OpenMP Offload Programming

Dr.-Ing. Michael Klemm
Chief Executive Officer
OpenMP Architecture Review Board

Agenda

- OpenMP Architecture Review Board
- Introduction to OpenMP Offload Features
- Case Study: NWChem TCE CCSD(T)
- Detachable Tasks

Introduction to OpenMP Offload Features

OpenMP

Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

Timing code (not needed, just to have a bit more code to show ☺)

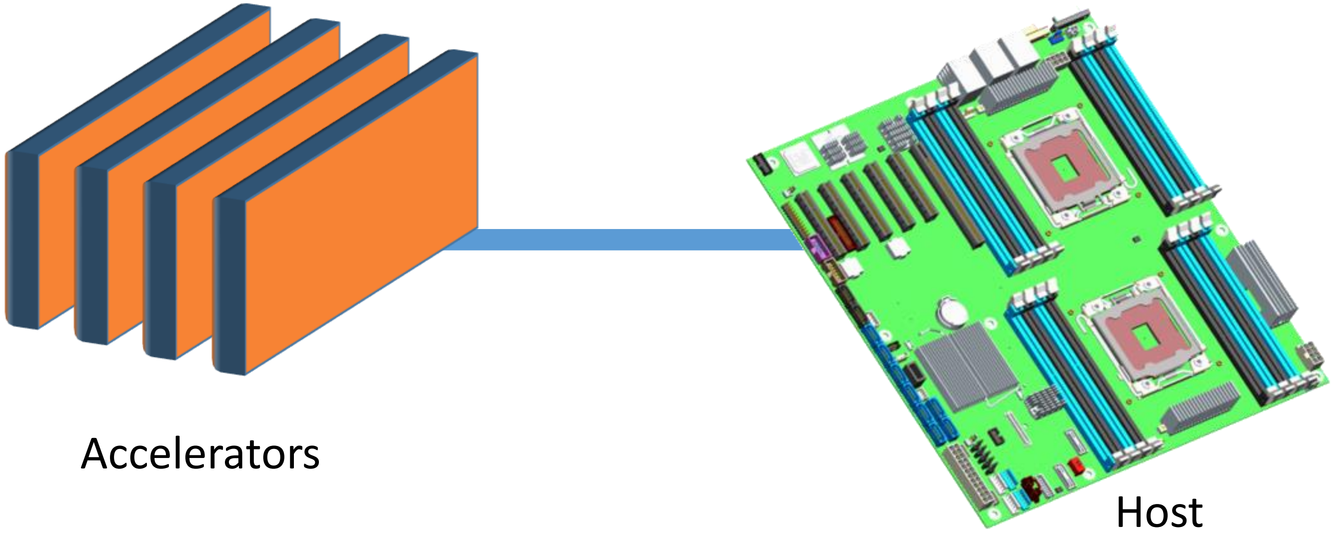
This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show ☺)

Don’t do this at home!
Use a BLAS library for this!

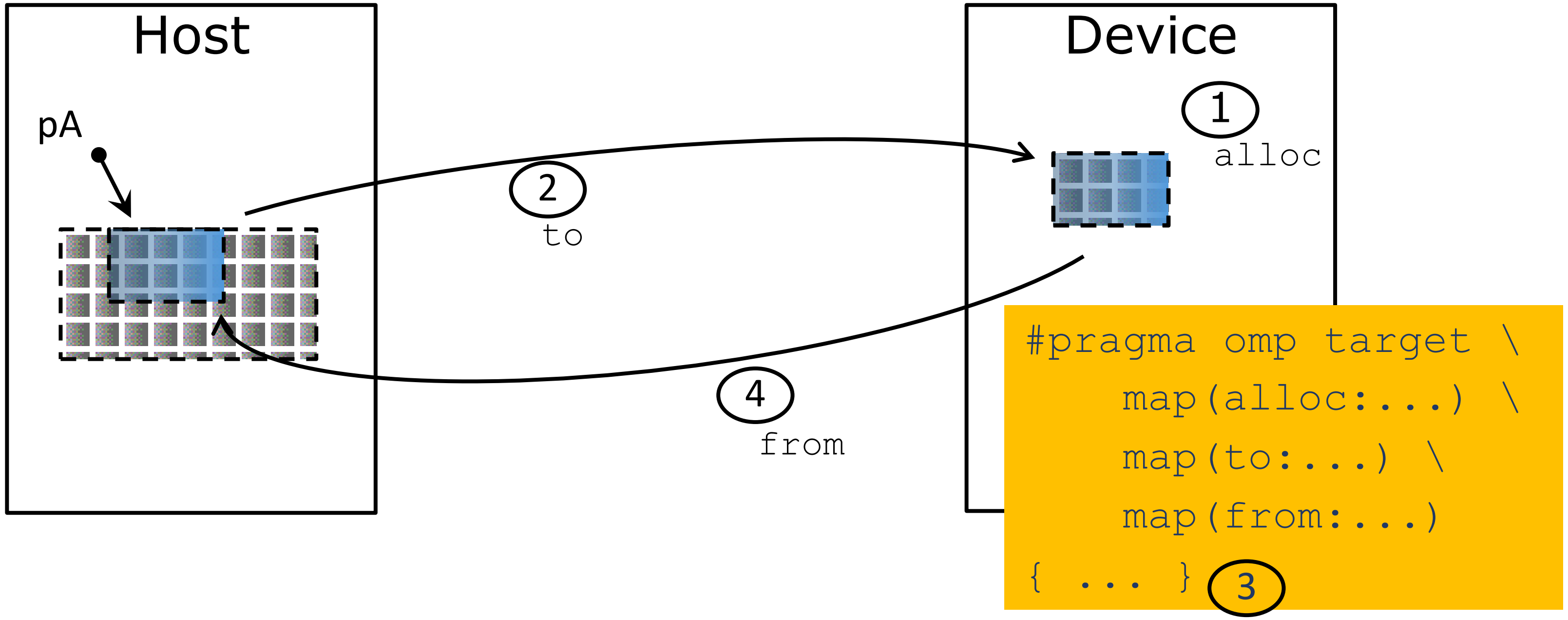
Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading



Execution Model

- Offload region and data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released



OpenMP for Devices - Constructs

- Transfer control and data from the host to the device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

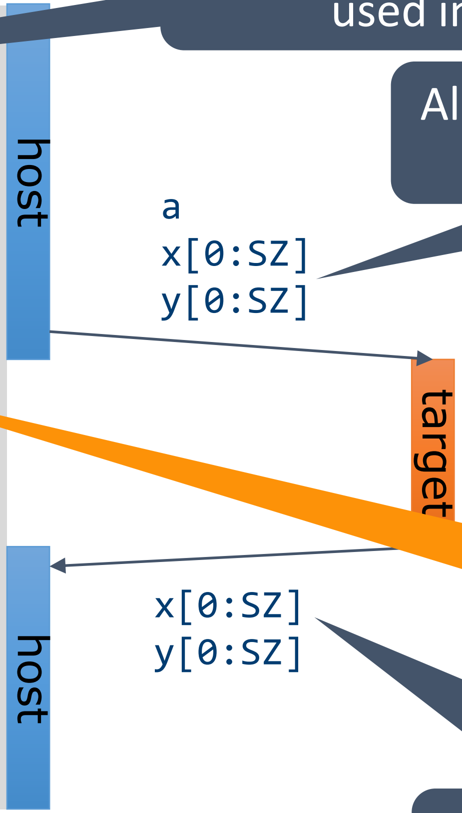
```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

Example: saxpy

```c
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target “map(tofrom:y[0:SZ])”
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler identifies variables that are used in the `target` region.

All accessed arrays are copied from host to device and back



Presence check: only transfer if not yet allocated on the device.

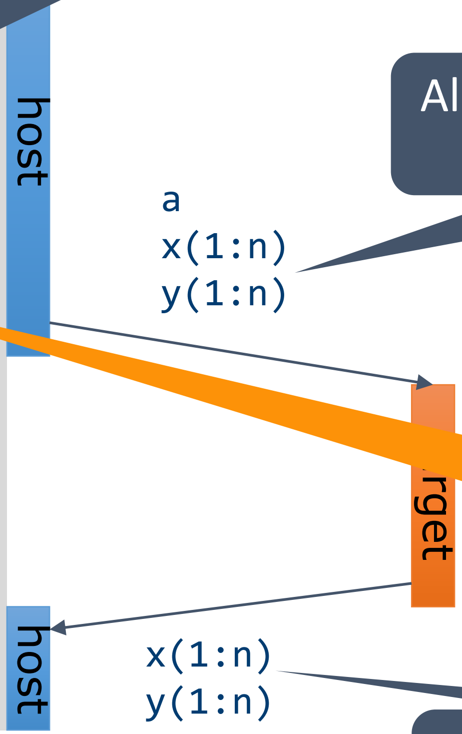
Copying x back is not necessary: it was not changed.

```
clang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
```

Example: saxpy

The compiler identifies variables that are used in the `target` region.

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

!$omp target “map(tofrom:y(1:n))”
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```



All accessed arrays are copied from host to device and back

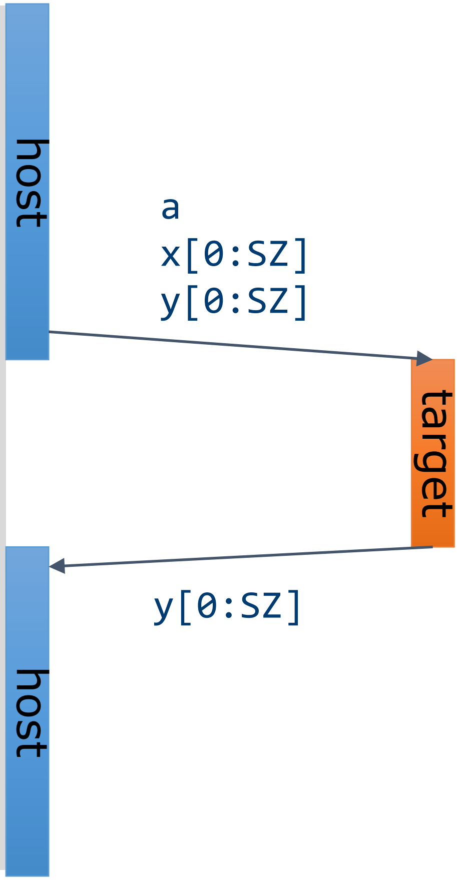
Presence check: only transfer if not yet allocated on the device.

Copying x back is not necessary: it was not changed.

```
flang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
```

Example: saxpy

```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:SZ]) \
                   map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

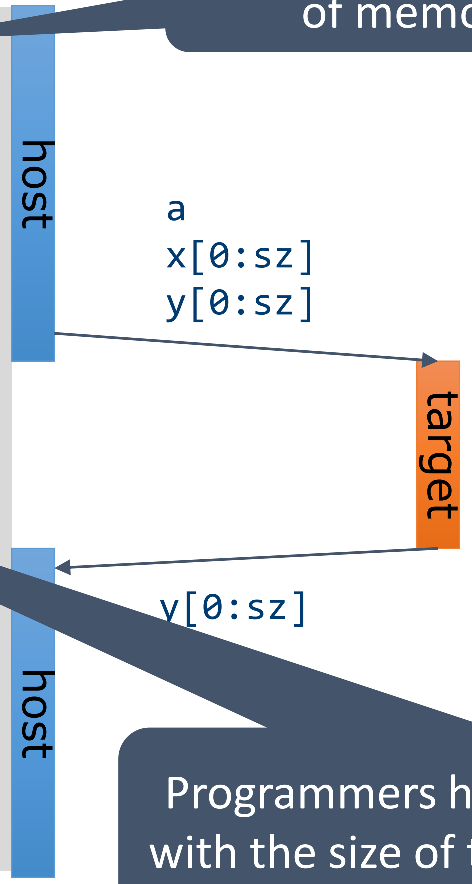



```
clang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
```

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler cannot determine the size of memory behind the pointer.



Programmers have to help the compiler with the size of the data transfer needed.

```
clang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
```

Creating Parallelism on the Target Device

- The `target` construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - This is intentional!

- OpenMP separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device
  - In theory, this can be combined with any OpenMP construct
  - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host

target

host

GPUs are multi-level devices: SIMD, threads, thread blocks

Create a team of threads to execute the loop in parallel using SIMD instructions.

```
clang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908
```

`teams` Construct

- Support multi-level parallel devices
- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Multi-level Parallel saxpy

- Manual code transformation
  - Tile the loops into an outer loop and an inner loop
  - Assign the outer loop to “teams” (OpenCL: work groups)
  - Assign the inner loop to the “threads” (OpenCL: work items)

Multi-level Parallel saxpy

- For convenience, OpenMP defines composite constructs to implement the required code transformations

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted
!$omp omp target teams distribute parallel do simd &
!$omp&           num_teams(num_blocks) map(to:x) map(tofrom:y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

Optimize Data Transfers

- Reduce the amount of time spent transferring data
  - Use map clauses to enforce direction of data transfer.
  - Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
void example() {
    float tmp[N], data_in[N], float data_out[N];
#pragma omp target data map(alloc:tmp[:N]) \
                        map(to:a[:N],b[:N]) \
                        map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
}   }
```

```
void zeros(float* a, int n) {
#pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}
```

```
void saxpy(float a, float* y, float* x, int n) {
#pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

`target data` Construct Syntax

- Create scoped data environment and transfer data from the host to the device and back
- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom | release | delete}:] list)
if(scalar-expr)
```

`target update` Construct Syntax

- Issue data transfers to or from existing data device environment
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

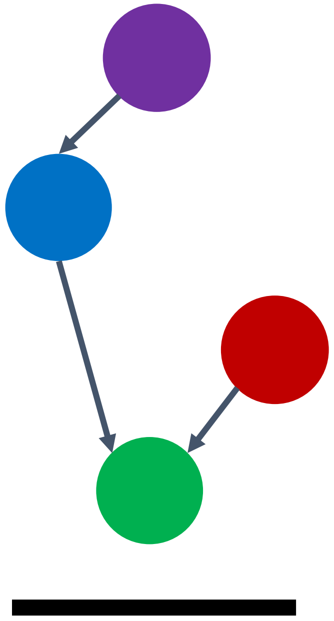
Example: `target data` and `target update`

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous (in OpenMP speak: they become an OpenMP task)

```
#pragma omp task                                                depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])  nowait      depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:z[:N])  nowait      depend(out:y)
    compute_3(b, z, N);

#pragma omp target map(to:y[:N]) map(to:z[:N])    nowait      depend(in:x) depend(in:y)
    compute_4(z, x, y, N);

#pragma omp taskwait
```



Case Study: NWChem TCE CCSD(T)

TCE: Tensor Contraction Engine
CCSD(T): Coupled-Cluster with Single, Double, and perturbative Triple replacements

NWChem

- Computational chemistry software package
  - Quantum chemistry
  - Molecular dynamics
- Designed for large-scale supercomputers
- Developed at the EMSL at PNNL
  - EMSL: Environmental Molecular Sciences Laboratory
  - PNNL: Pacific Northwest National Lab
- URL: http://www.nwchem-sw.org

Finding Offload Candidates

- Requirements for offload candidates
  - Compute-intensive code regions (kernels)
  - Highly parallel
  - Compute scaling stronger than data transfer, e.g., compute $O(n^3)$ vs. data size $O(n^2)$

Example Kernel (1 of 27 in total)

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                  h7d,triplesx,t2sub,v2sub)
c     Declarations omitted.
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h3d*h2d
       triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1   - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

1.5GB data transferred (host to device)

1.5GB data transferred (device to host)

- All kernels have the same structure
- 7 perfectly nested loops
- Some kernels contain inner product loop (then, 6 perfectly nested loops)
- Trip count per loop is equal to “tile size” (20-30 in production)
- Naïve data allocation (tile size 24)
  - Per-array transfer for each `target` construct
  - triplesx: 1458 MB
  - t2sub, v2sub: 2.5 MB each

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data alloc(triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p5 4d,h7,triplesx,t2sub,v2sub)
          end if
c         same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
        end do
        do ...
c         Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
        end do
        call sum_energy(energy, triplesx)
      end do
!$omp target exit data release(triplesx(1:size))
```

Allocate 1.5GB data once, stays on device.

Update 2x2.5MB of data for (potentially) multiple kernels.

- Reduced data transfers:
  - triplesx:
    - allocated once
    - always kept on the target
  - t2sub, v2sub:
    - allocated after comm.
    - kept for (multiple) kernel invocations

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data alloc(triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p5 ... 4d,h7,triplesx
          end if
c         same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
        end do
        do ...
c         Similar structure for sd_t_d2_1 until sd_t_d2_9, inc
        end do
        call sum_energy(energy, triplesx)
      end do
!$omp target exit data release(triplesx(1:size))
```

Allocate 1.5G ... stays on

Update 2x2.5 ... (potentially) r

```
      subroutine sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1               h7d,triplesx,t2sub,v2sub)
c     Declarations omitted.
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h3d*
       triplesx(h2h
     1   - t2sub(h7
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

Presence check determines that arrays have been allocated in the device data environment already.

Advanced Task Synchronization

Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - HIP, CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: CUDA memory transfers

```
do_something();
cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
do_something_else();
cudaStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completions events with task execution?

Try 1: Use just OpenMP Tasks

```
void cuda_example() {
#pragma omp task      // task A
    {
        do_something();
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
    }
    #pragma omp task // task B
    {
        do_something_else();
    }
    #pragma omp task // task C
    {
        cudaStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Race condition between the tasks A & C, task C may start execution before task A enqueues memory transfer.

- This solution does not work!

Try 2: Use just OpenMP Tasks Dependences

```
void cuda_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
    }
    #pragma omp task                   // task B
    {
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        cudaStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Synchronize execution of tasks through dependence. May work, but task C will be blocked waiting for the data transfer to finish

- This solution may work, but
  - takes a thread away from execution while the system is handling the data transfer.
  - may be problematic if called interface is not thread-safe

OpenMP Detachable Tasks

- OpenMP 5.0 introduces the concept of a detachable task
  - Task can detach from executing thread without being “completed”
  - Regular task synchronization mechanisms can be applied to await completion of a detached task
  - Runtime API to complete a task

- Detached task events: `omp_event_t` datatype
- Detached task clause: `detach(event)`
- Runtime API: `void omp_fulfill_event(omp_event_t *event)`

Detaching Tasks

```
omp_event_t *event;
void detach_example() {
#pragma omp task detach(event)
    {
        important_code();
    } ①

    #pragma omp taskwait ② ④
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. `taskwait` construct cannot complete
3. Signal event for completion
4. Task completes and `taskwait` can continue

Putting It All Together

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {
 ③ omp_fulfill_event((omp_event_t *) cb_data);
}
void cuda_example() {
    omp_event_t *cuda_event;
#pragma omp task detach(cuda_event) // task A
    {
        do_something();
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
        cudaStreamAddCallback(stream, callback, cuda_event, 0);
 ①  }
#pragma omp task                   // task B
        do_something_else();

#pragma omp taskwait②④
#pragma omp task                   // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

Removing the `taskwait` Construct

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {
  ② omp_fulfill_event((omp_event_t *) cb_data);
}
void cuda_example() {
    omp_event_t *cuda_event;
#pragma omp task depend(out:dst) detach(cuda_event) // task A
    {
        do_something();
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
  ①     cudaStreamAddCallback(stream, callback, cuda_event, 0);
    }
#pragma omp task                      // task B
        do_something_else();

#pragma omp task depend(in:dst)    ③  // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

Summary

- OpenMP API is ready to use Intel discrete GPUs for offloading compute
  - Mature offload model w/ support for asynchronous offload/transfer
  - Tightly integrates with OpenMP multi-threading on the host
- More, advanced features (not covered here)
  - Memory management API
  - Interoperability with native data management
  - Interoperability with native streaming interfaces
  - Unified shared memory support

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information

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

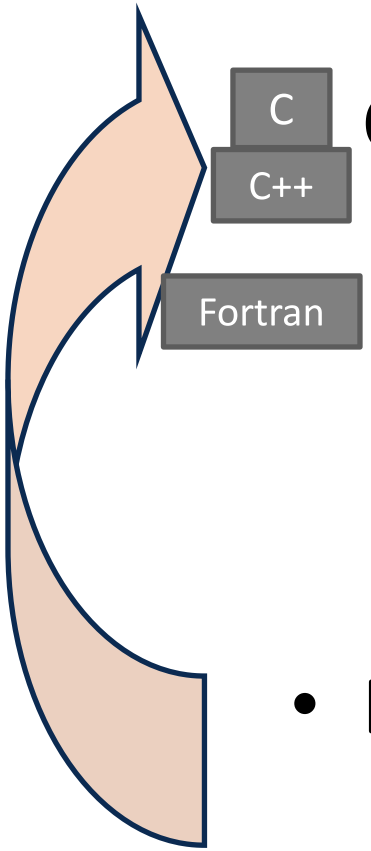
ThreadSanitizer: Overview

- 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)
```

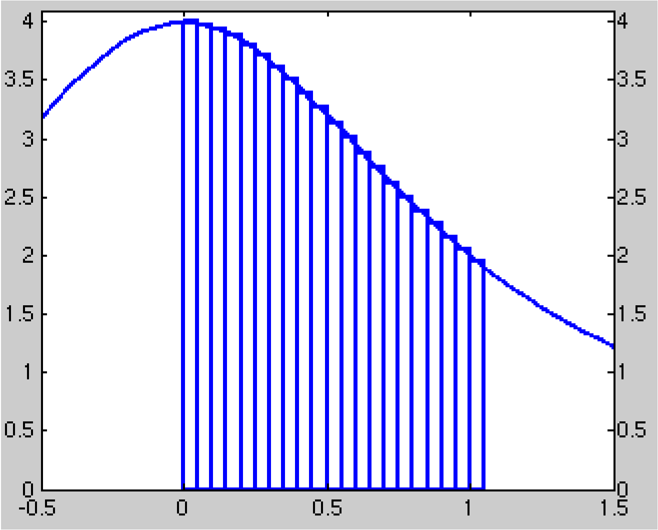
Intel Inspector XE

- 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;
}
```

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



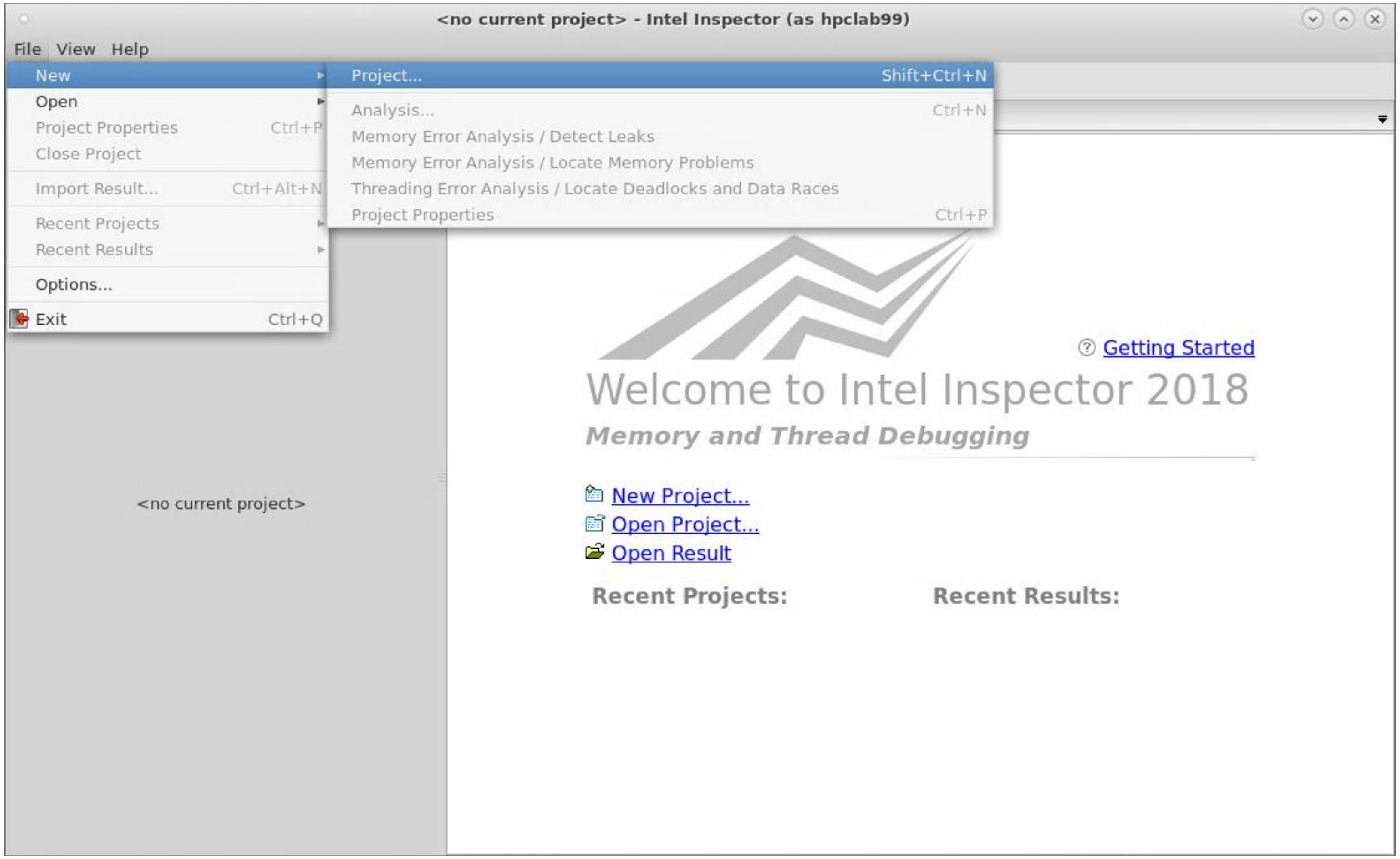
PI example / 2

```
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?

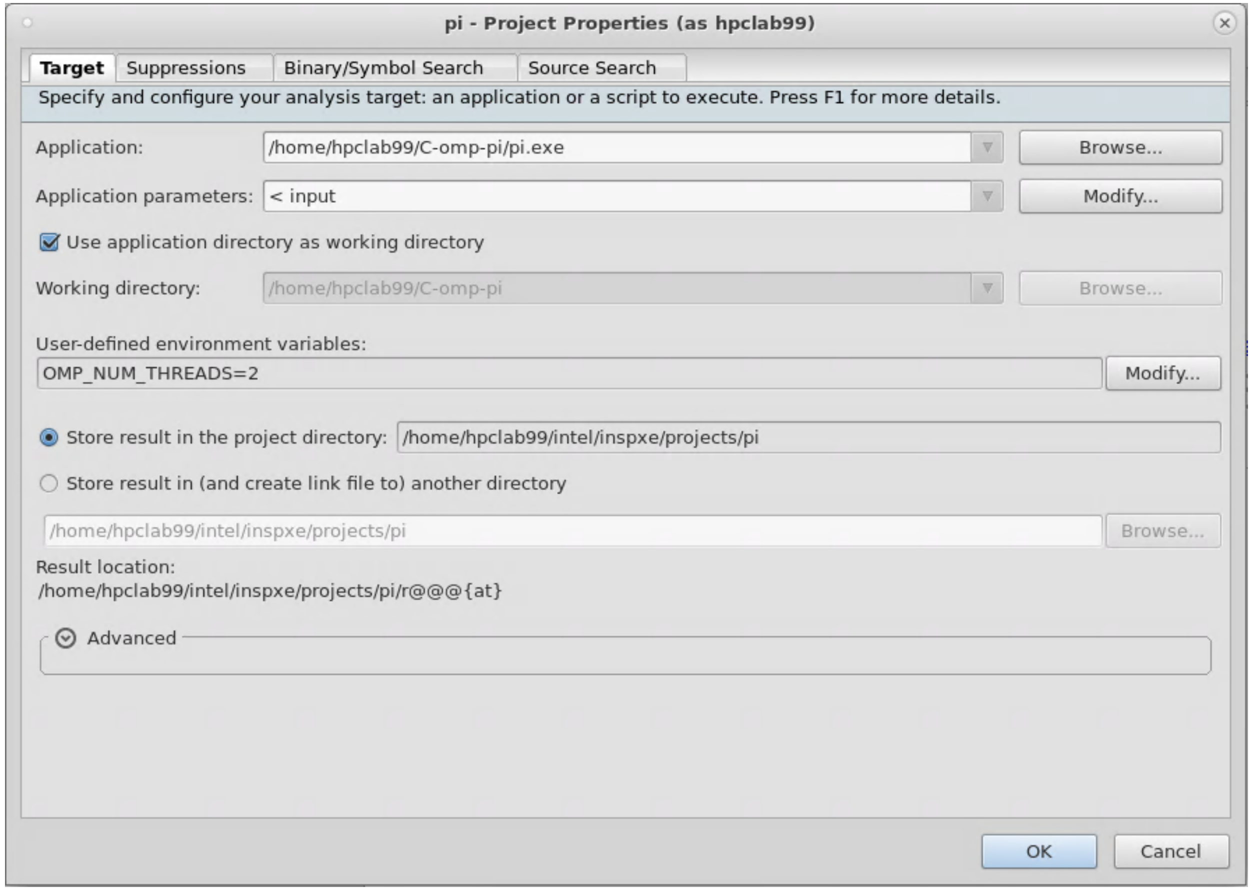
Inspector XE: create project / 1

```
$ 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

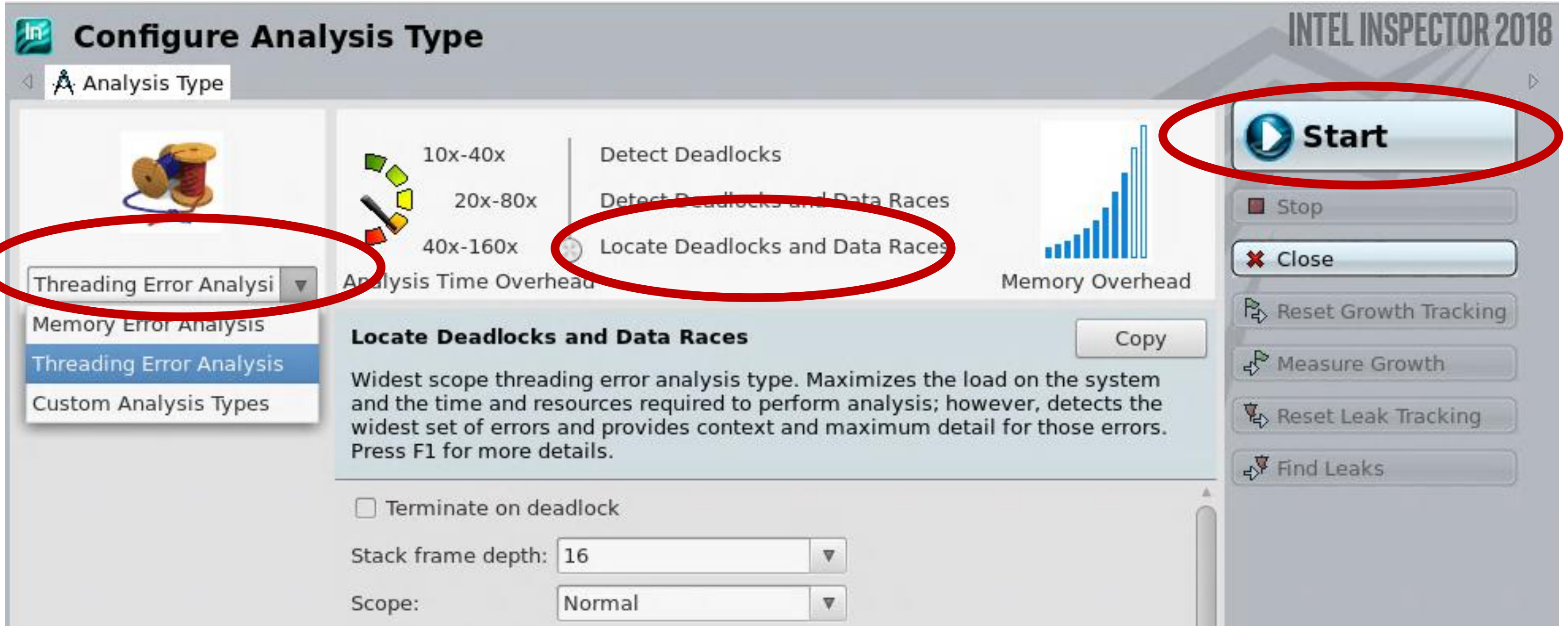


Inspector XE: configure analysis

Threading Error Analysis Modes

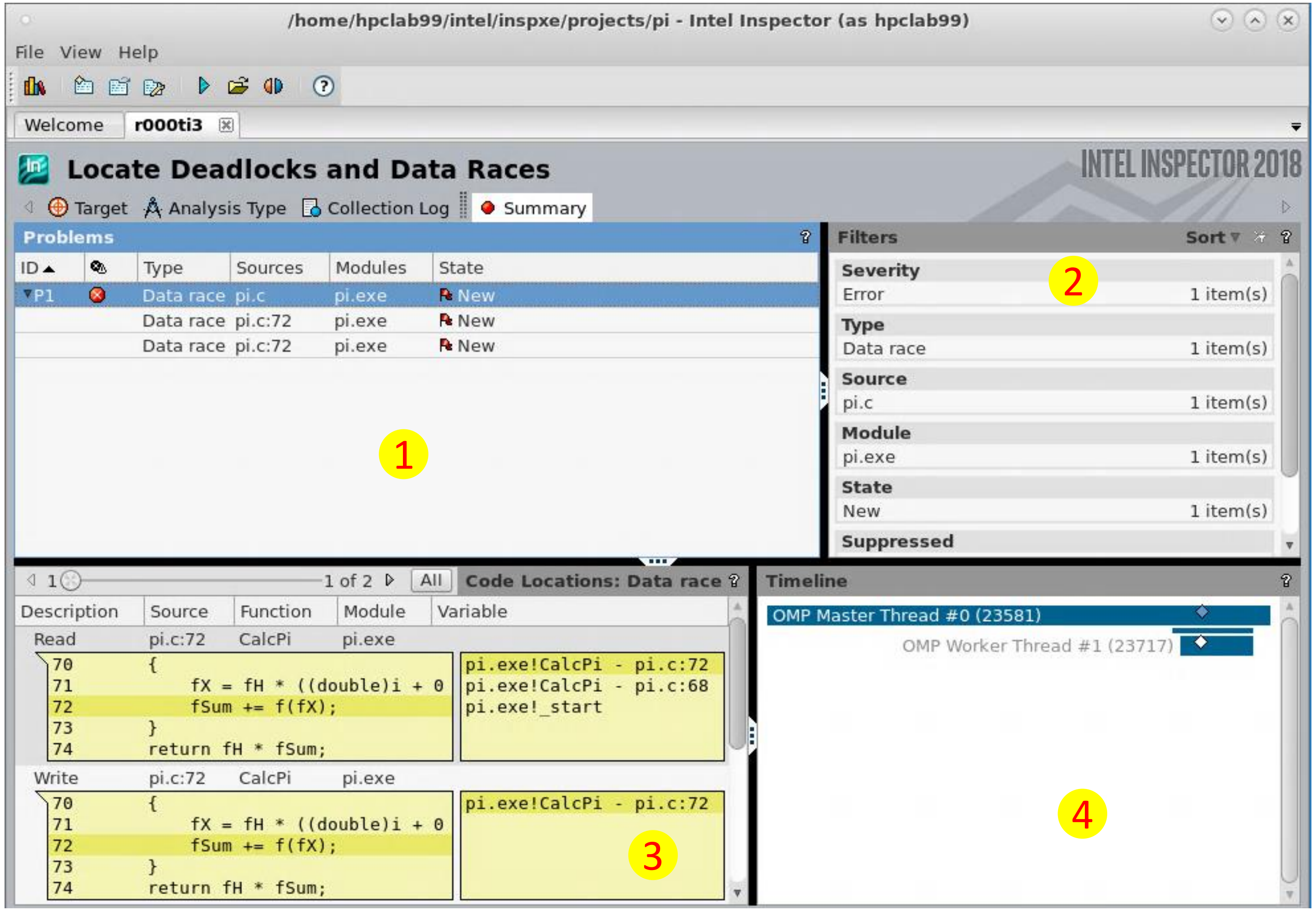
1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead



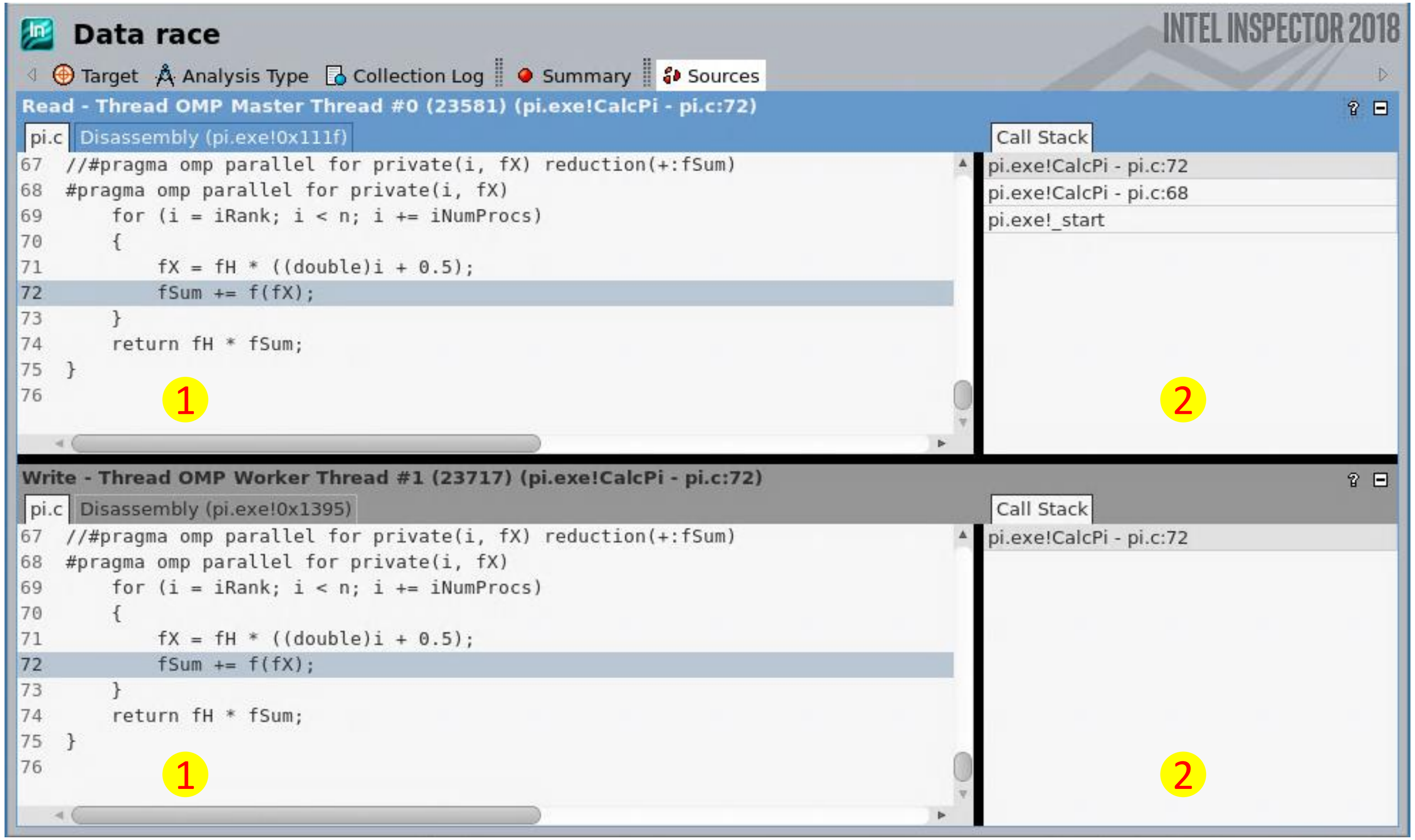
Inspector XE: results / 1

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

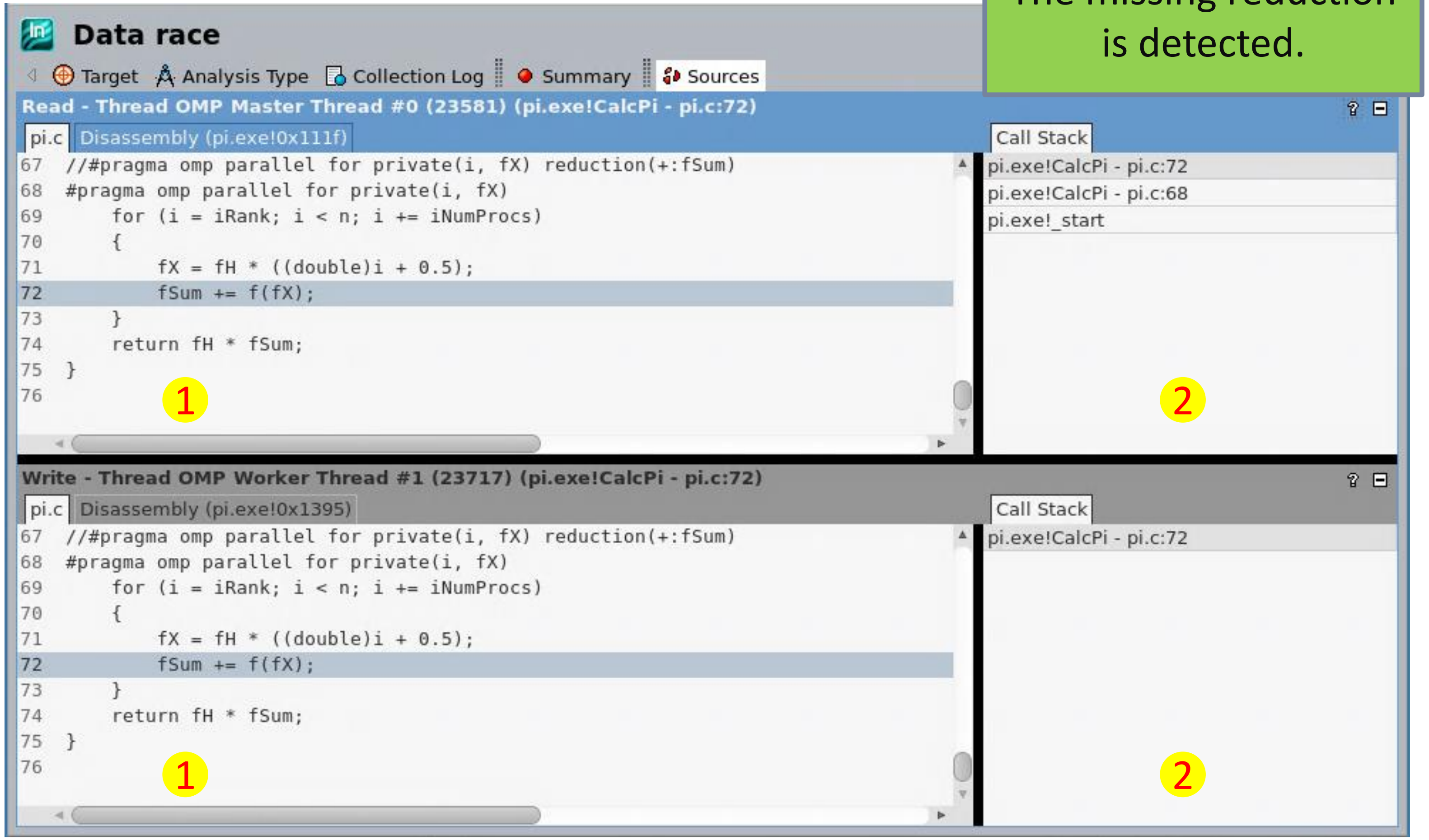


Inspector XE: results / 3

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

2 Corresponding Call Stack

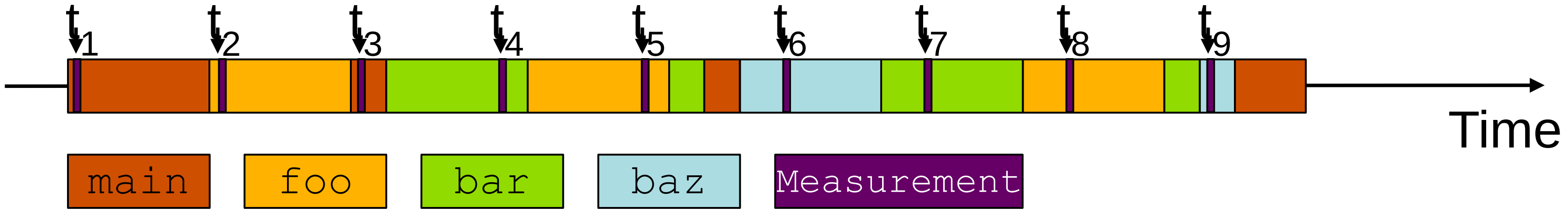
The missing reduction is detected.



Sampling vs. Instrumentation

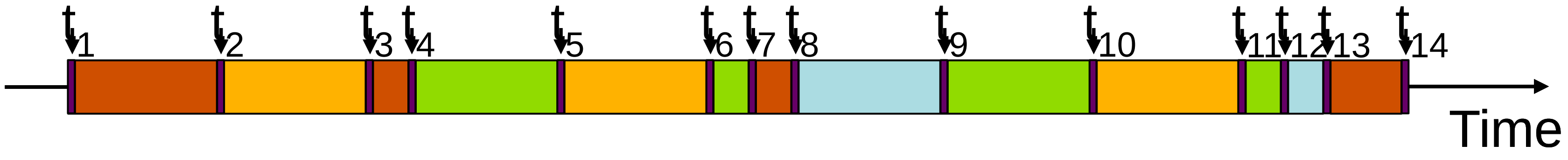
Sampling

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



Instrumentation

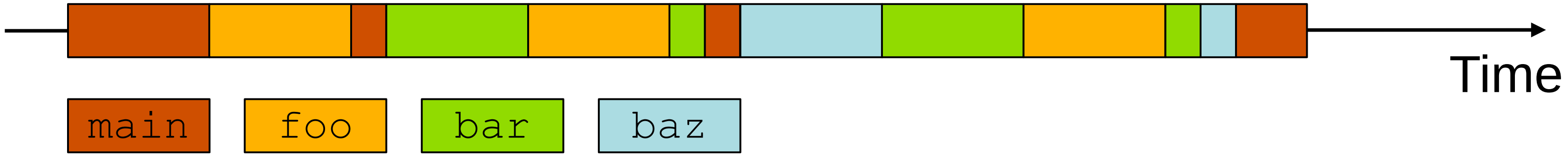
- Every event of interest is captured directly
- More detailed and *exact* information
- Typically: recompile for instrumentation



Tracing vs. Profiling

Trace

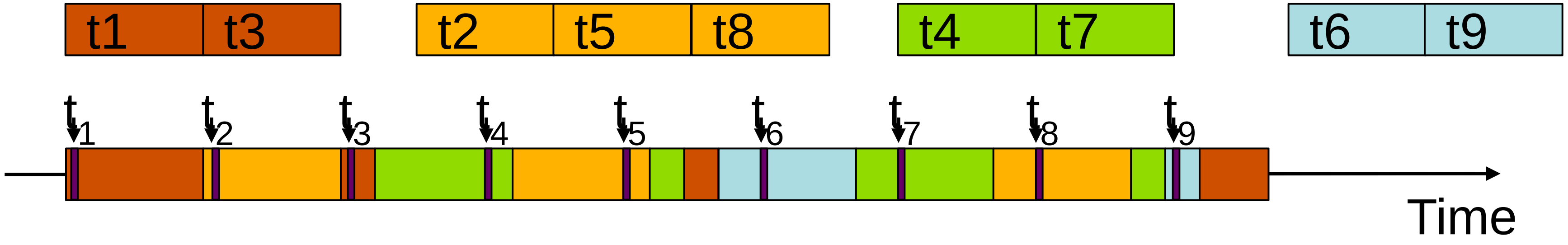
- Chronologically ordered sequence of event records



Profile from instrumentation

- Aggregated information

Profile from sampling

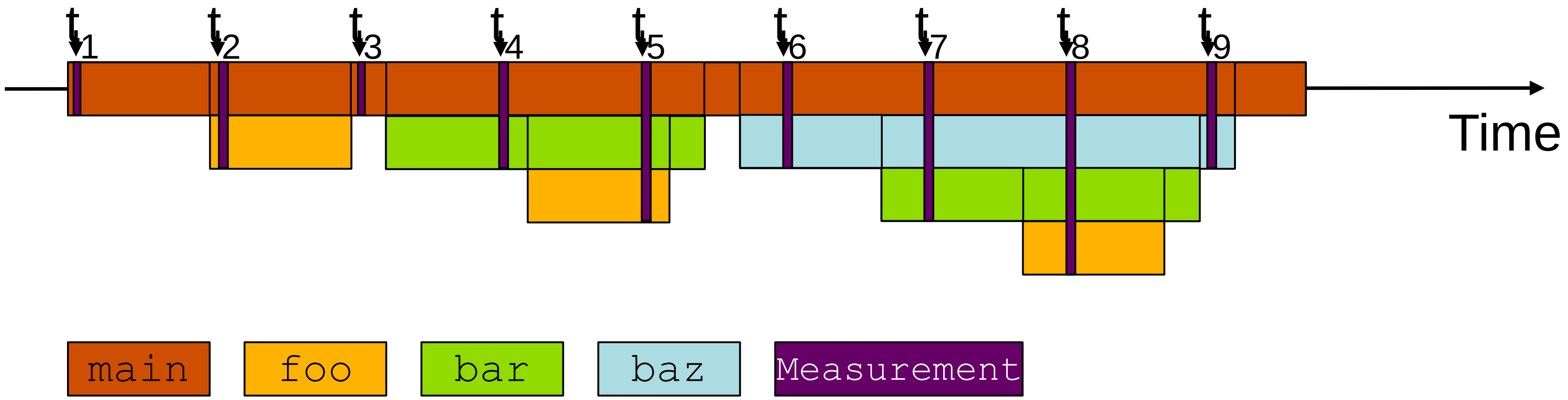


OMPT support for sampling

- 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

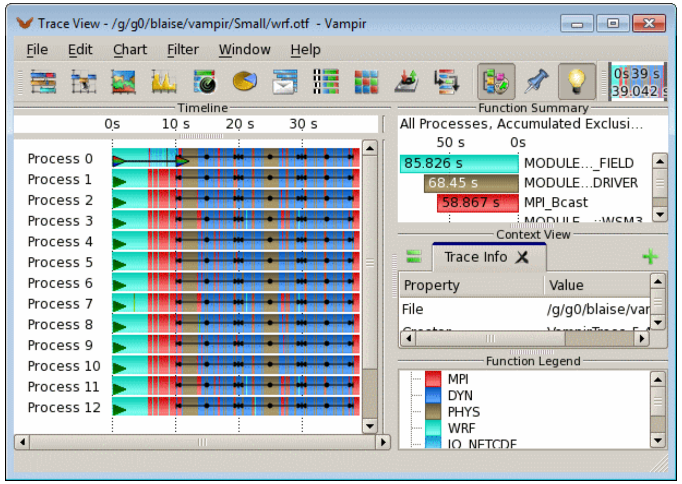
Data processing

- Scalasca : trace-based analysis

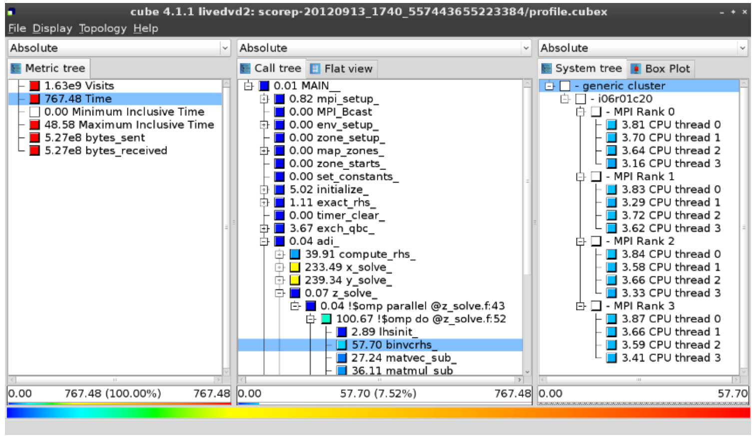
Data presentation

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

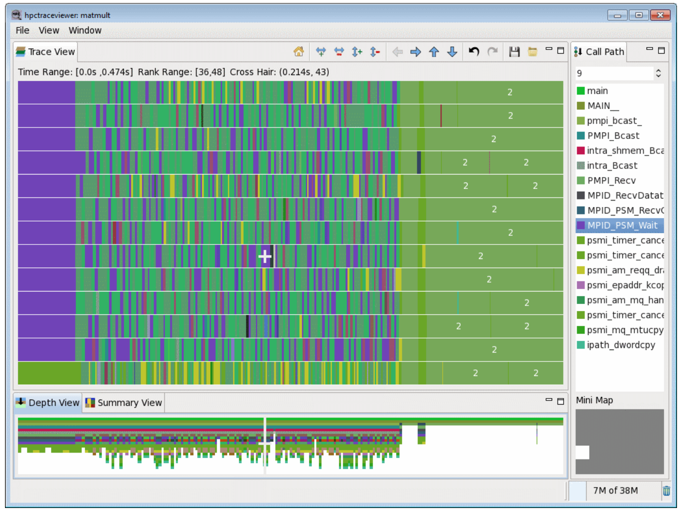
Performance tools GUI



VAMPIR



cube scalasca



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

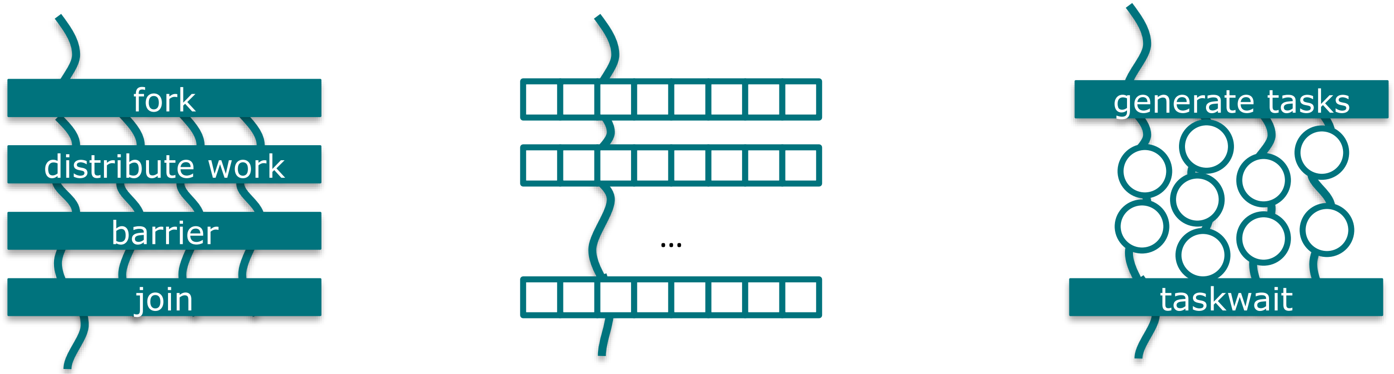
`loop` Construct

- 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

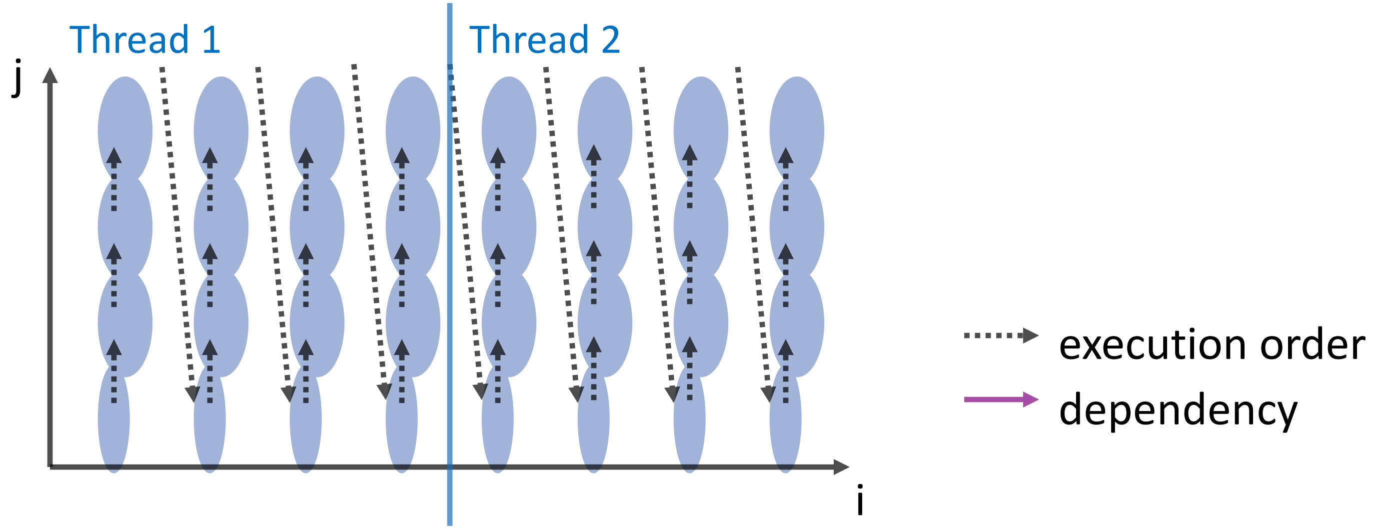
- Loop-carried dependency:
  - → Loop iterations depend on each other
  - → Source of dependency must scheduled before sink of the dependency

- 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!)

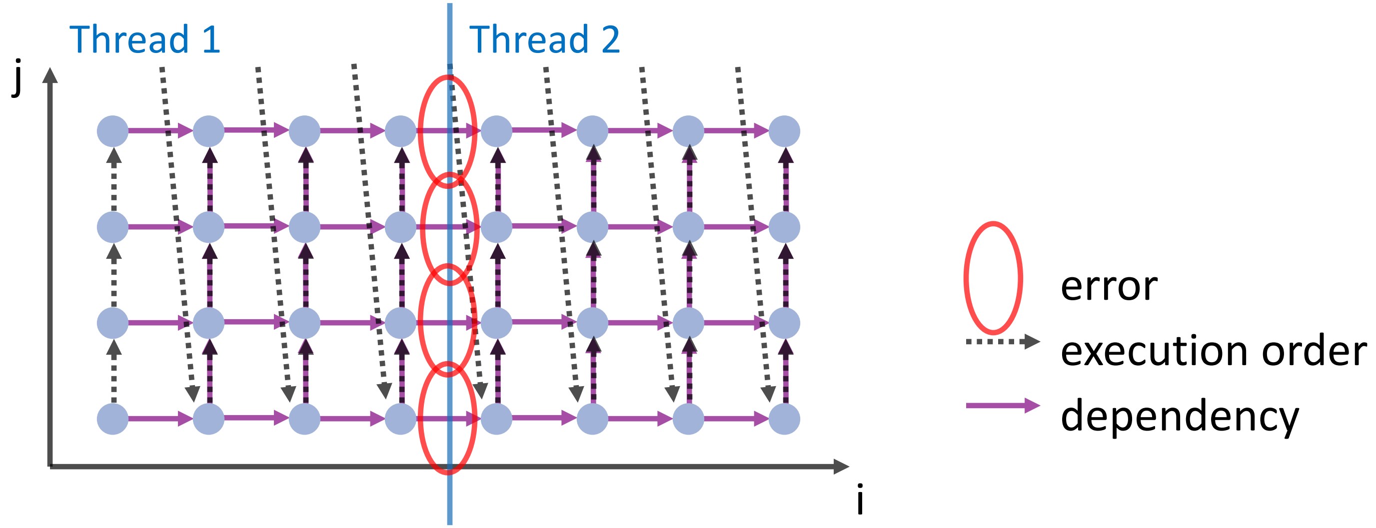
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i][j],
                    b[i][j], a[i][j]);
    }
}
```



Non-parallelizable Loops

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

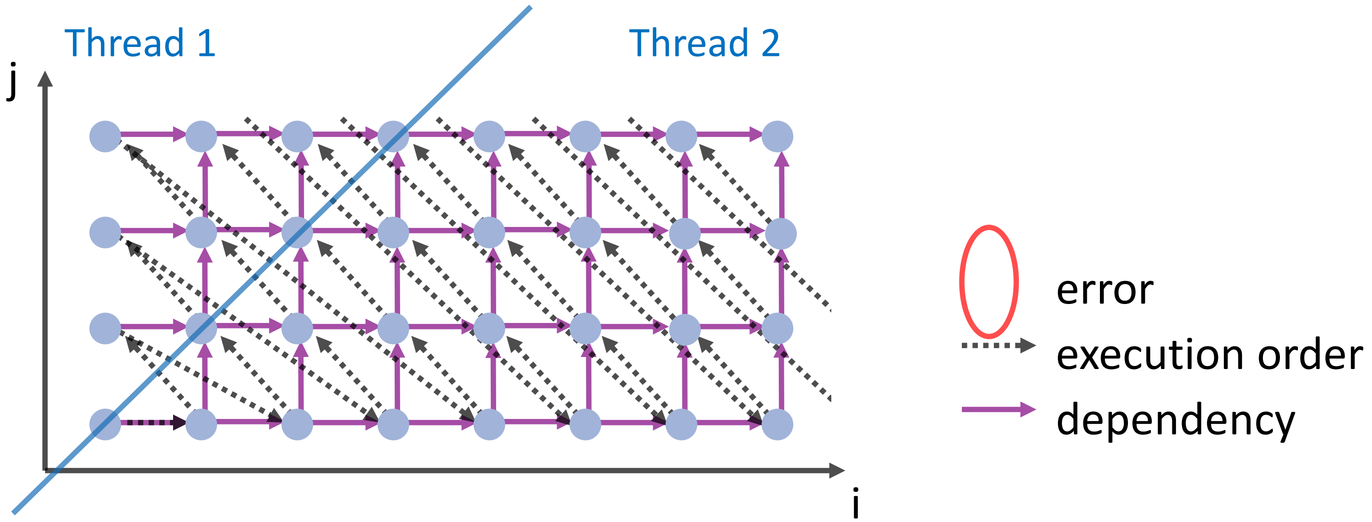
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
}
```



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 Meta-Programming

The `metadirective` Directive

- Construct OpenMP directives for different OpenMP contexts
- Limited form of meta-programming for OpenMP directives and clauses

```
#pragma omp target map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
            when( device={arch(nvptx)}: teams loop ) \
            default( parallel loop )
for (i = lb; i < ub; i++)
   v3[i] = v1[i] * v2[i];
```

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: target ) &
            default( target map(mapper(vec_map),tofrom: vec) )
!$omp teams distribute simd
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end teams distribute simd
!$omp end metadirective
```

Nothing Directive

The nothing Directive

- The `nothing` directive makes meta programming a bit clearer and more flexible.
- If a certain criterion matches, the nothing directive can stand to indicate that no (other) OpenMP directive should be used.
  - → The `nothing` directive is implicitly added if no condition matches

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: &
                  target teams distribute parallel do simd) &
            default( nothing )
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end metadirective
```

Error Directive

`Error` Directive Syntax

- Syntax (C/C++)

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

- Syntax (Fortran)

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

- Clauses

one of: `at(compilation)`, `at(runtime)`

one of: `severity(fatal)`, `severity(warning)`

`message(`*`msg-string`*`)`

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.

```
#pragma omp parallel
{
    if (omp_get_num_threads() % 2) {
#pragma omp error at(runtime) severity(warning) \
                  message(“Running on odd number of threads\n”);
    }
    do_stuff_that_works_best_with_even_thread_count();
}
```

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.
- More useful in combination with OpenMP metadirective

```
!$omp begin metadirective &
            when( arch={fancy_processor}: parallel ) &
            default( error severity(fatal) at(compilation) &
                               message(“No implementation available” )
    call fancy_impl_for_fancy_processor()
!$omp end metadirective
```

OpenMP API Version 5.1
State of the Union

OpenMP

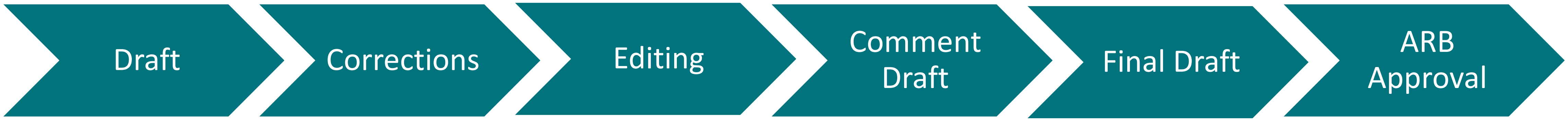
Architecture Review Board

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

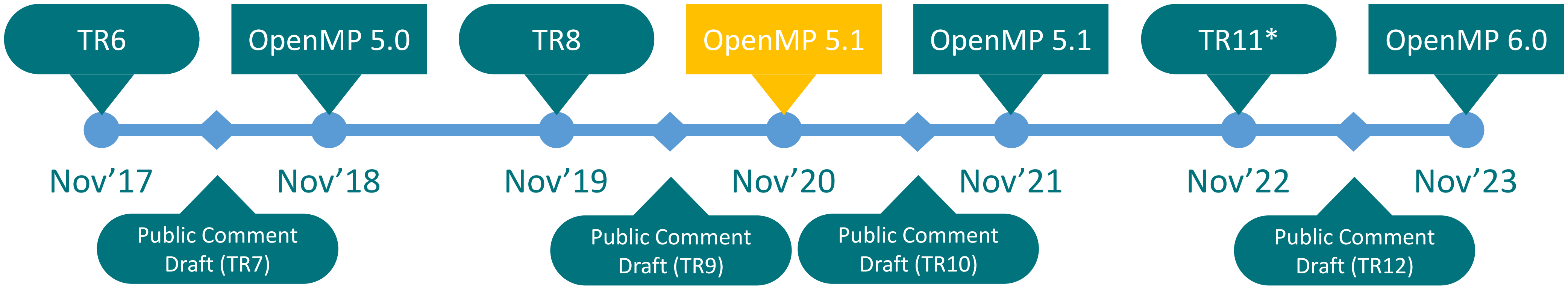
- Modifications of the OpenMP specification follow a (strict) process:

- 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

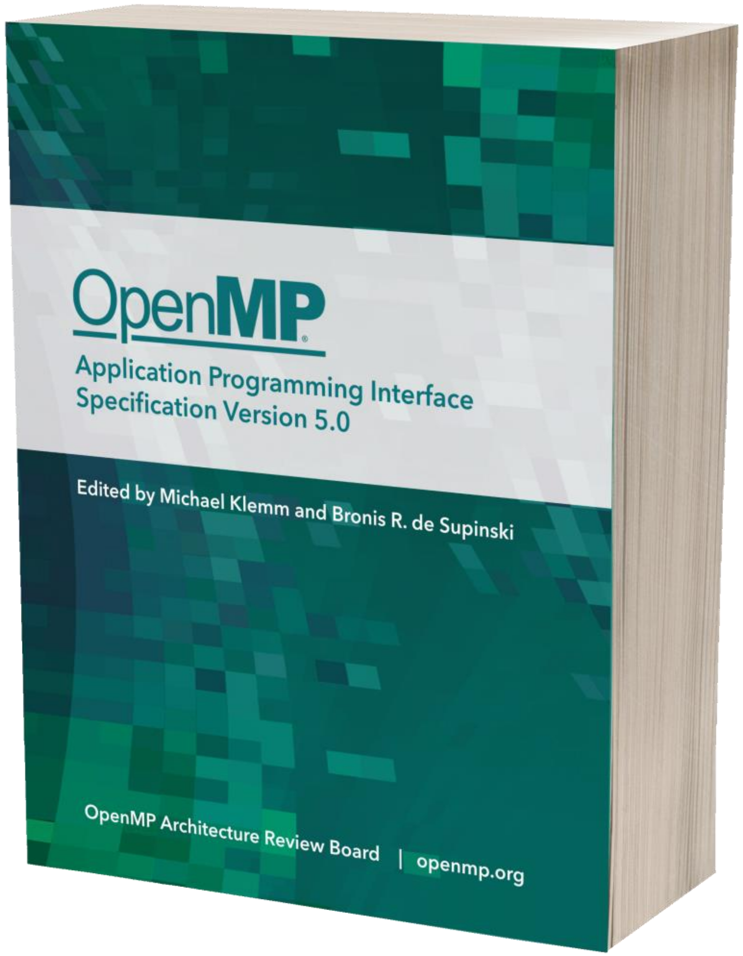


* Numbers assigned to TRs may change if additional TRs are released.

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

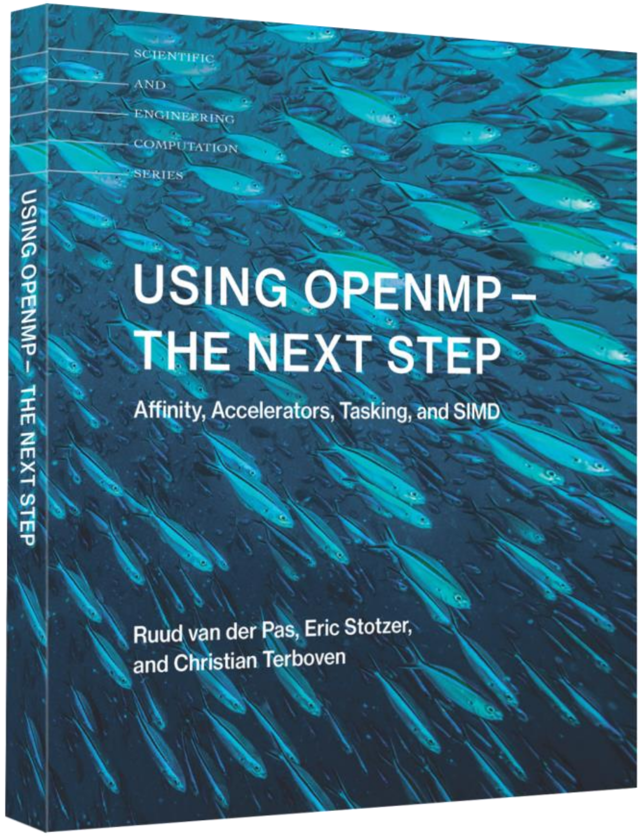
Printed OpenMP API Specification



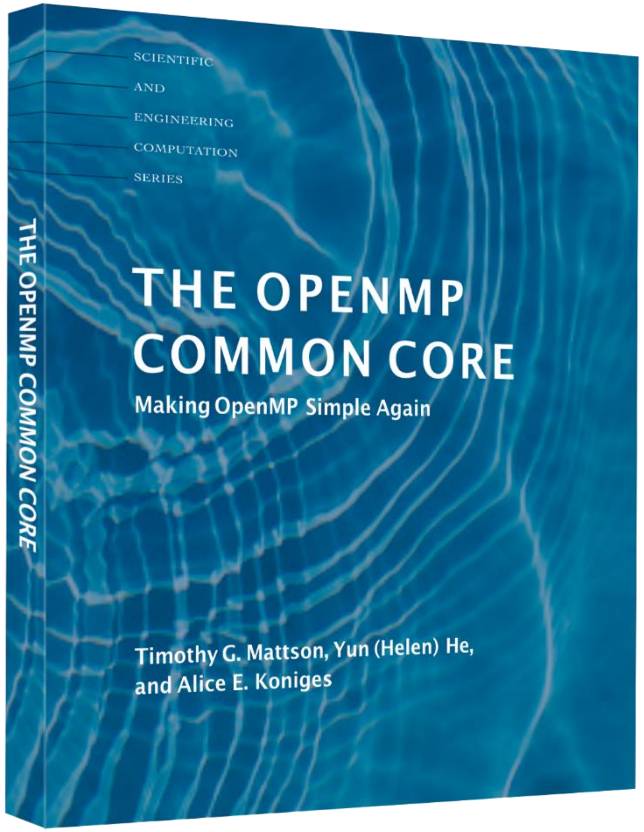
- Save your printer-ink and get the full specification as a paperback book!
  - Always have the spec in easy reach.
  - Includes the entire specification with the same pagination and line numbers as the PDF.
  - Available at a near-wholesale price.

- 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