OpenMP

OpenMP Offload Programming

OpenMP

Introduction to OpenMP Offload Features

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 ☺)

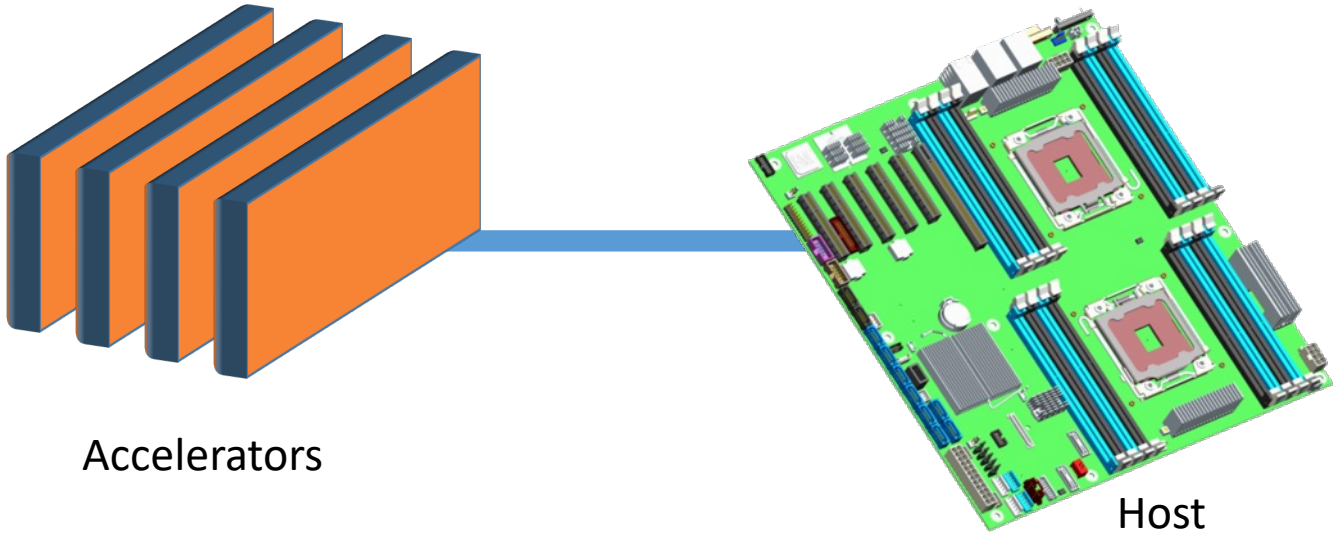
This is the code we want to execute on a target device (i.e., GPU)

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!

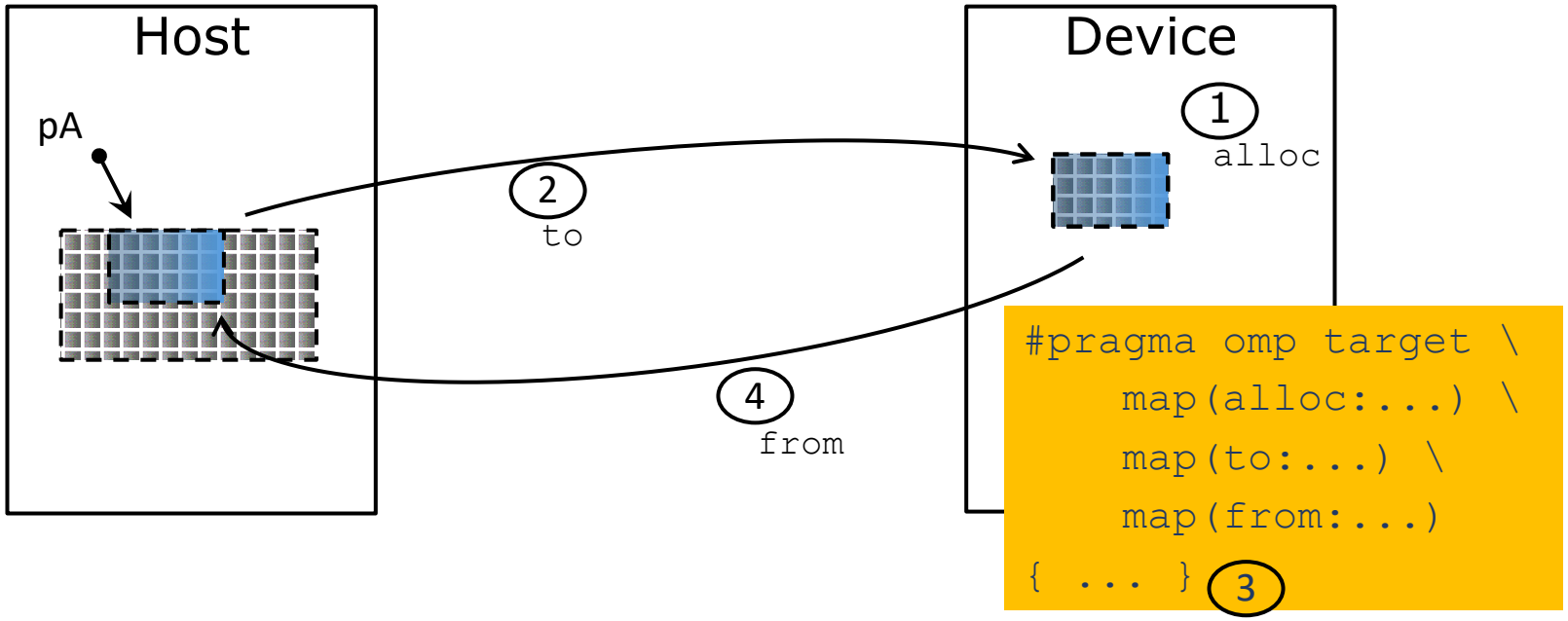
Device Model

- 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

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)

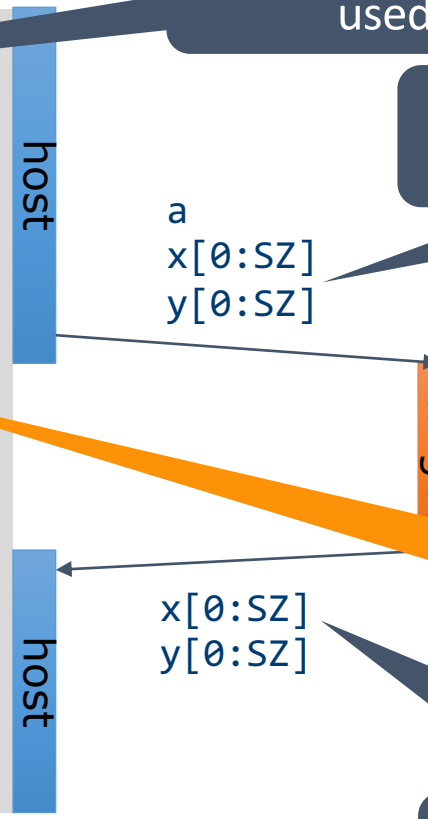
```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

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

Example: saxpy

```
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

a
x[0:SZ]
y[0:SZ]

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

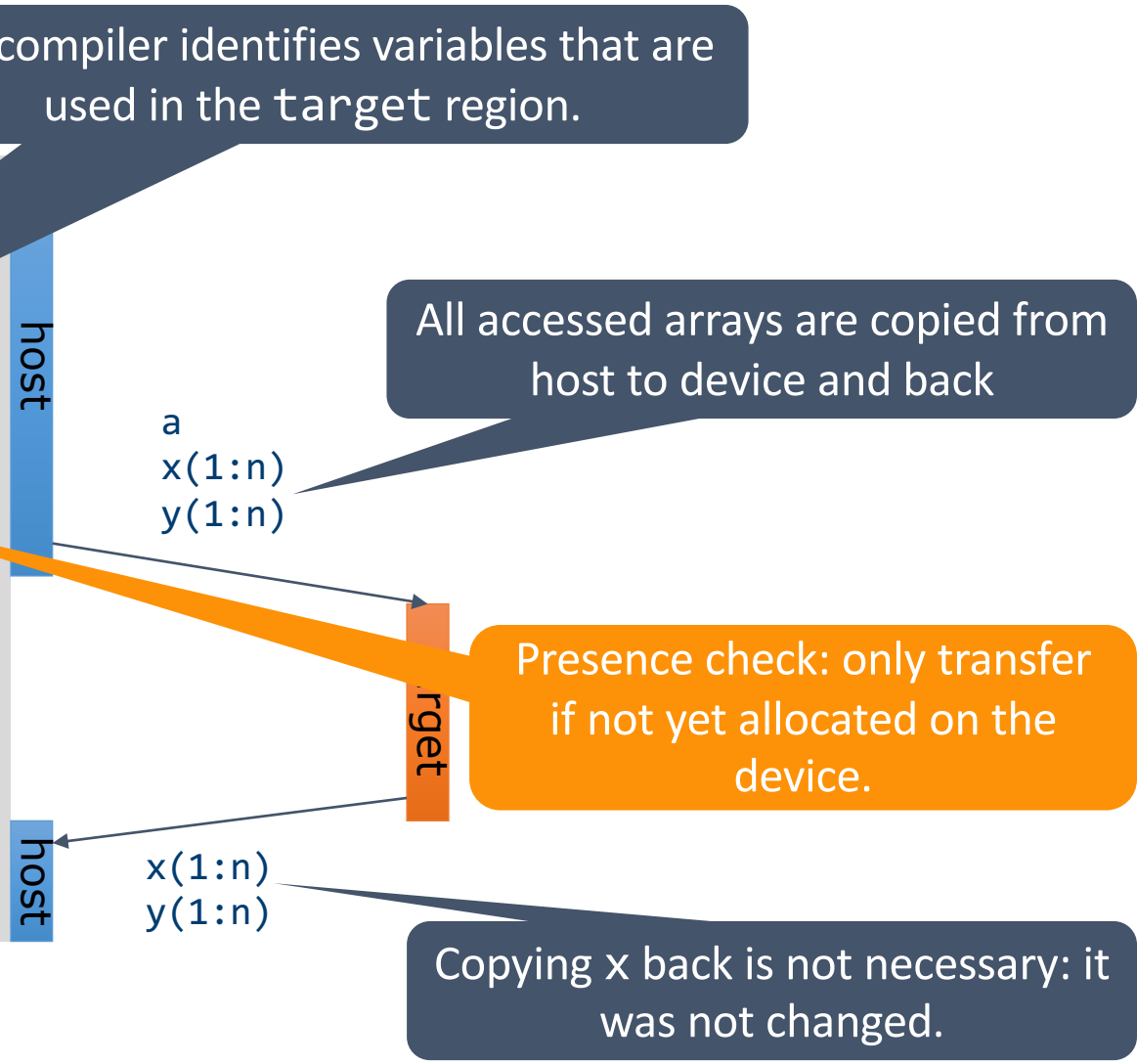
x[0:SZ]
y[0:SZ]

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

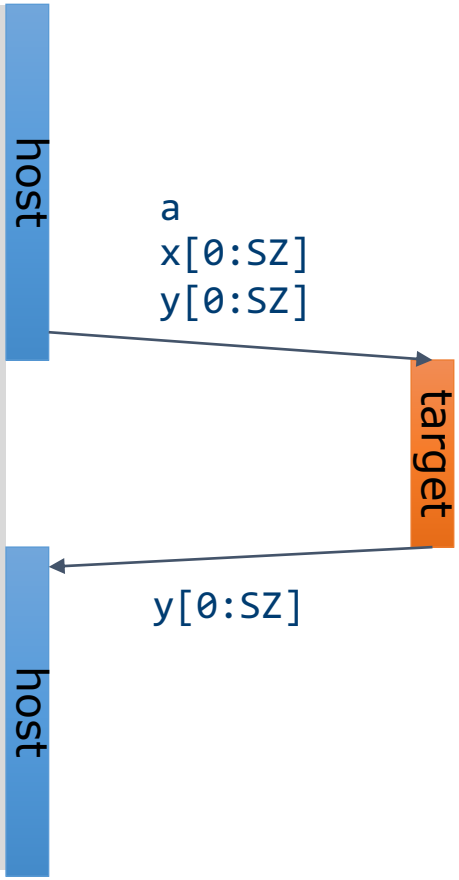
```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

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




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

Example: saxpy

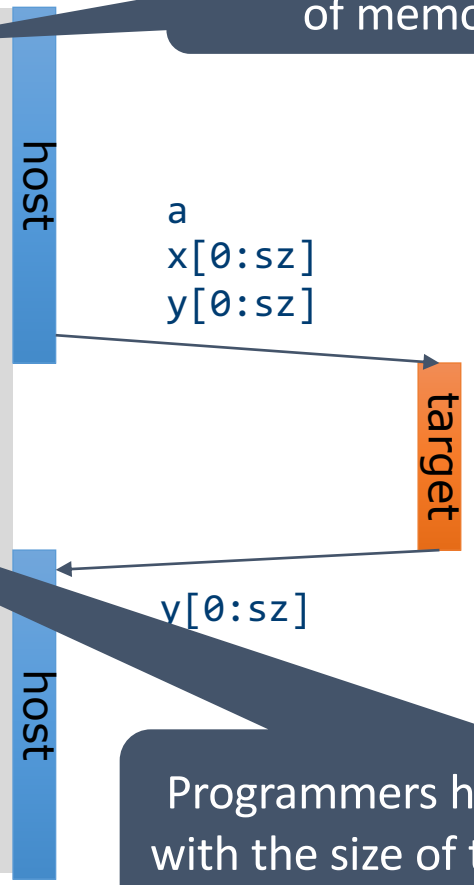
```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:SZ]) \
                   map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;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

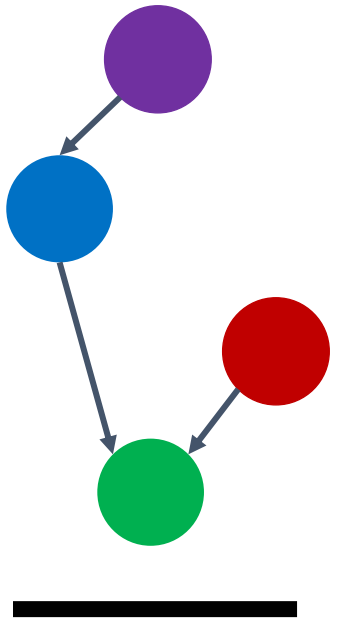
Example: `target data` and `target update`

```
#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
```



OpenMP

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

Detaching Tasks

```
omp_event_handle_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) {
  ③omp_fulfill_event(*((omp_event_handle_t *) cb_data));
}
void cuda_example() {
    omp_event_handle_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_handle_t *) cb_data));
}
void cuda_example() {
    omp_event_handle_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)         // task C
    {                              ③
        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

Visit www.openmp.org for more information

Programming OpenMP

Tools for OpenMP Programming

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

OpenMP

OpenMP Tools

- Correctness Tools
  - → ThreadSanitizer
  - → Intel Inspector XE (or whatever the current name is)
- Performance Analysis
  - → Performance Analysis basics
  - → Overview on available tools
  - → Case Study: CG

Correctness 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

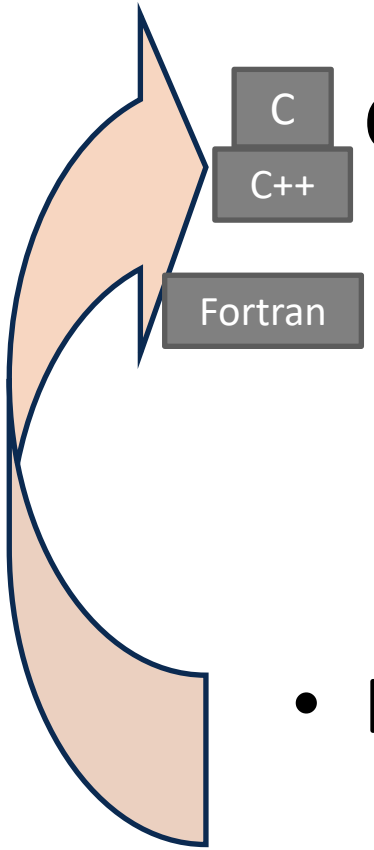
ThreadSanitizer: Overview

- 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

```
 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)
```

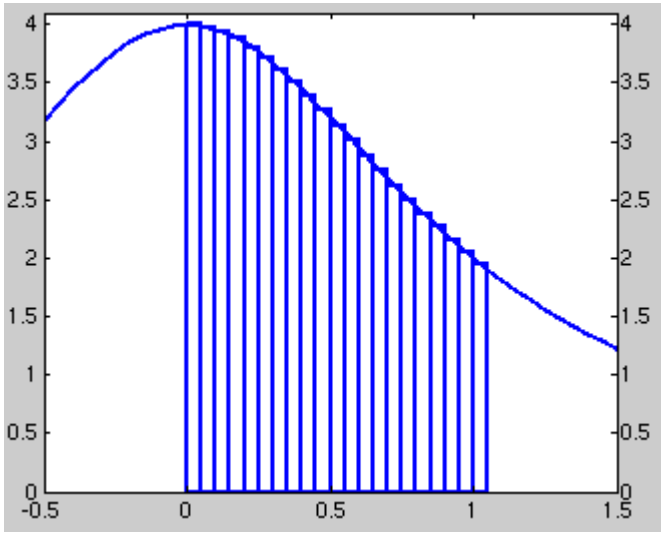
Intel Inspector XE

- 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;
}
```

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



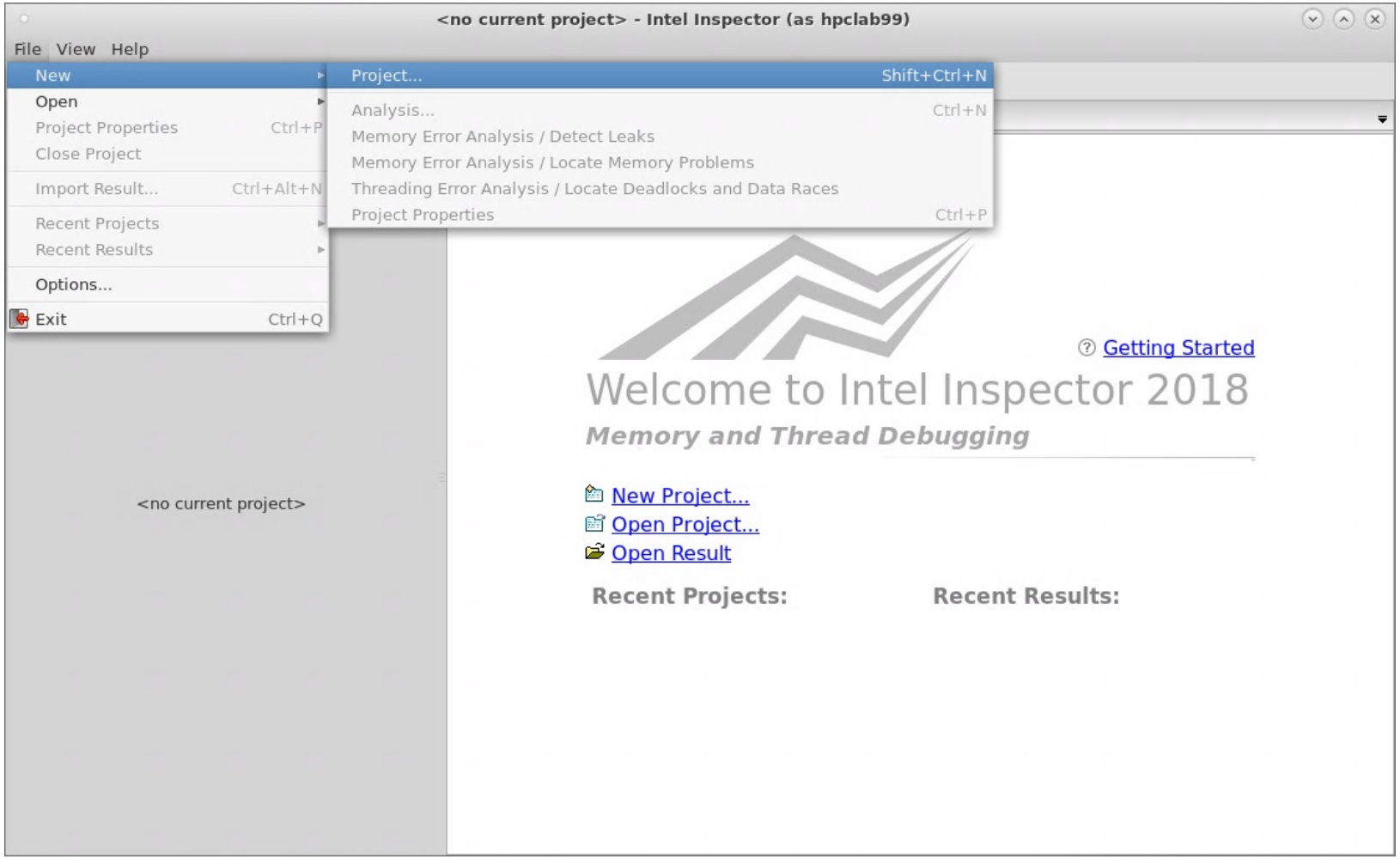
PI example / 2

```
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?

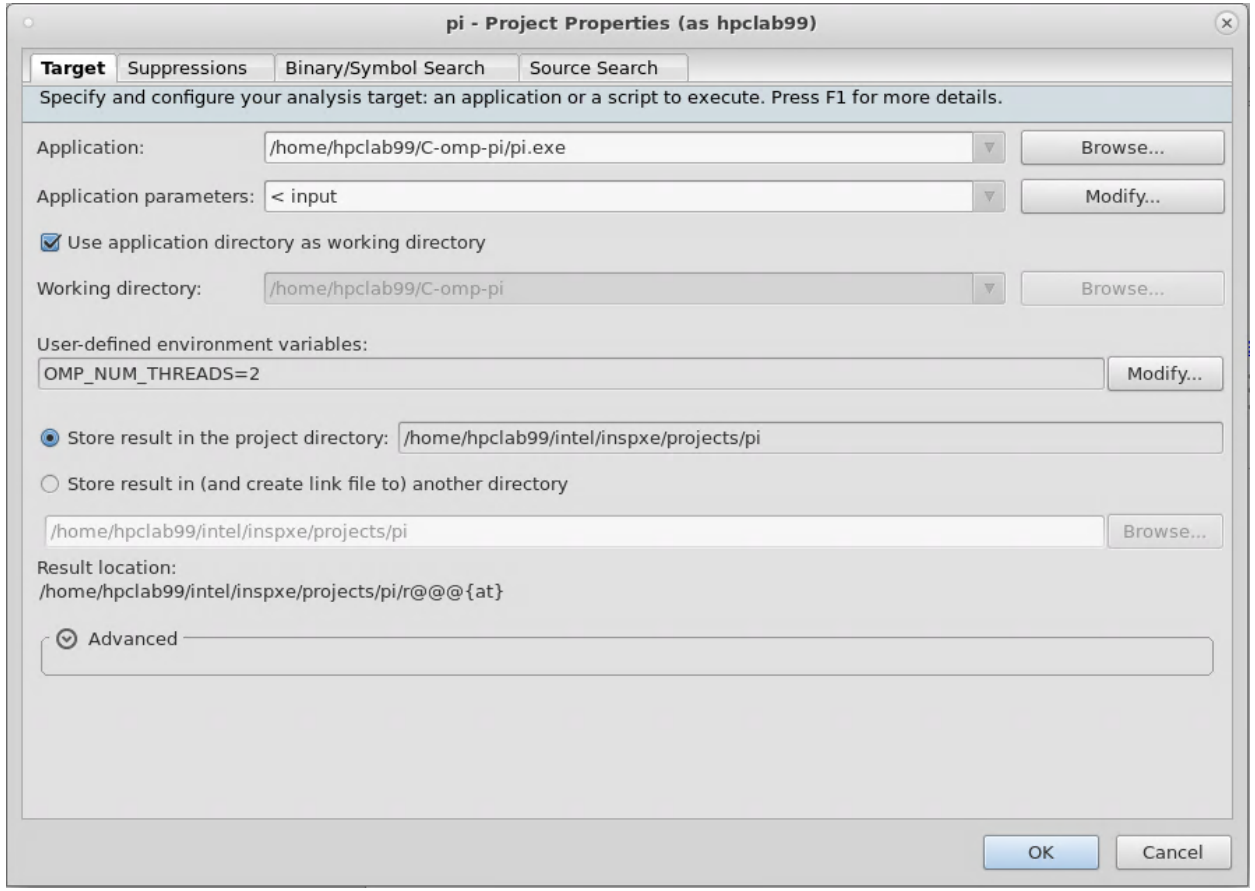
Inspector XE: create project / 1

```
$ 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

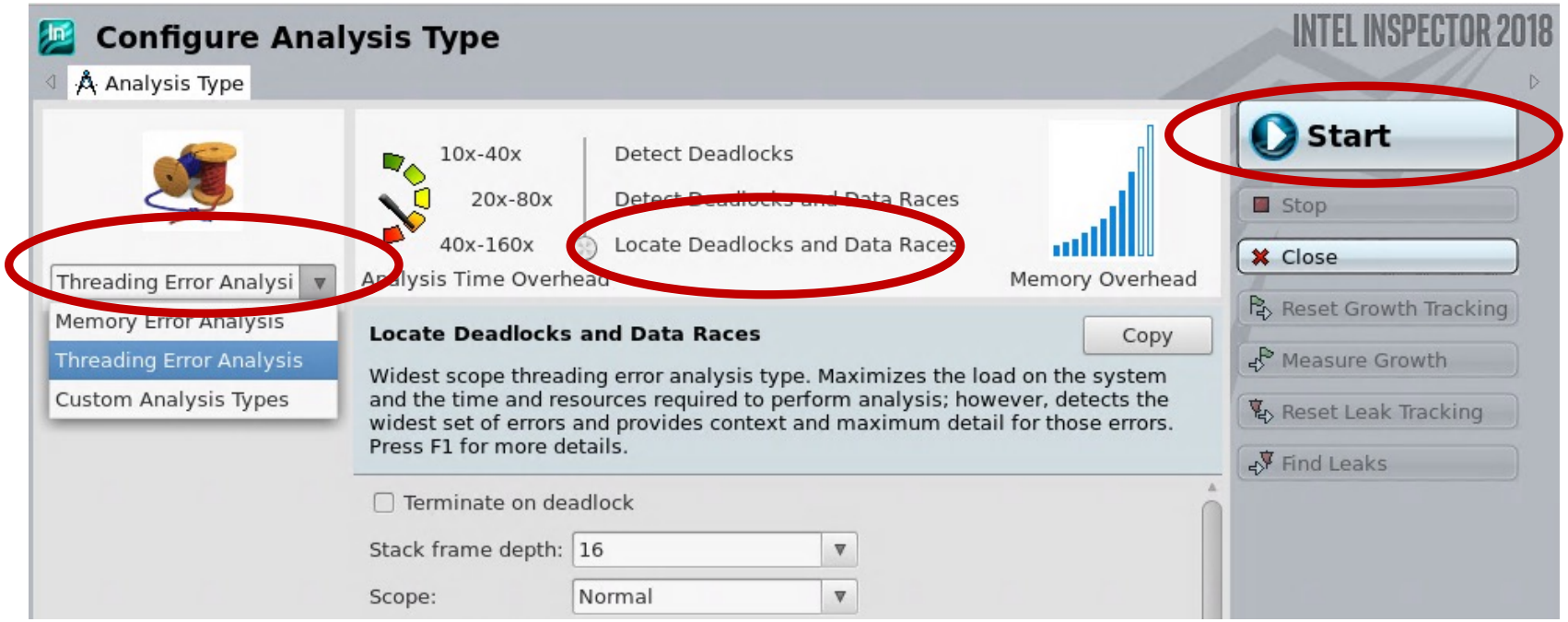


Inspector XE: configure analysis

Threading Error Analysis Modes

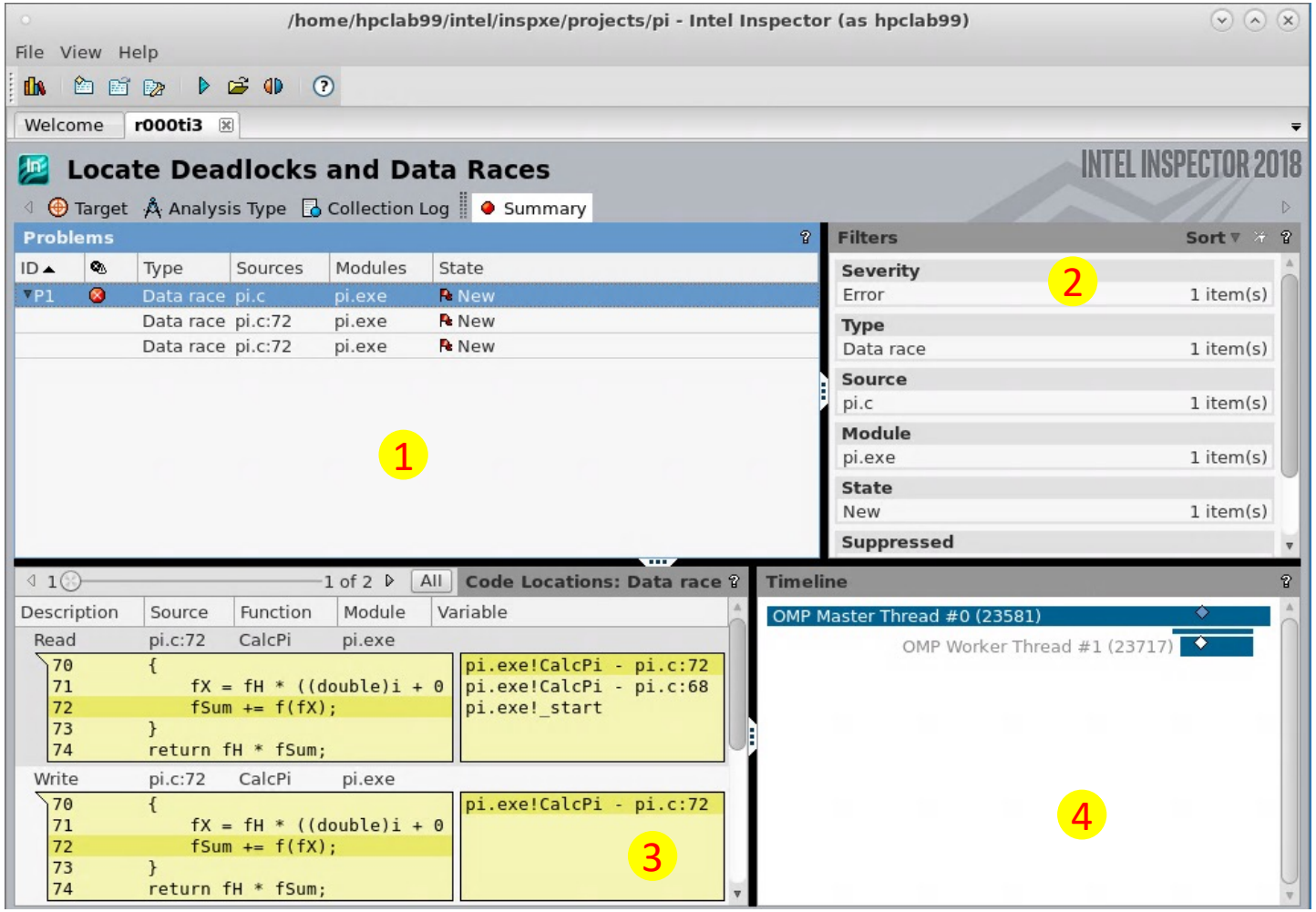
1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead



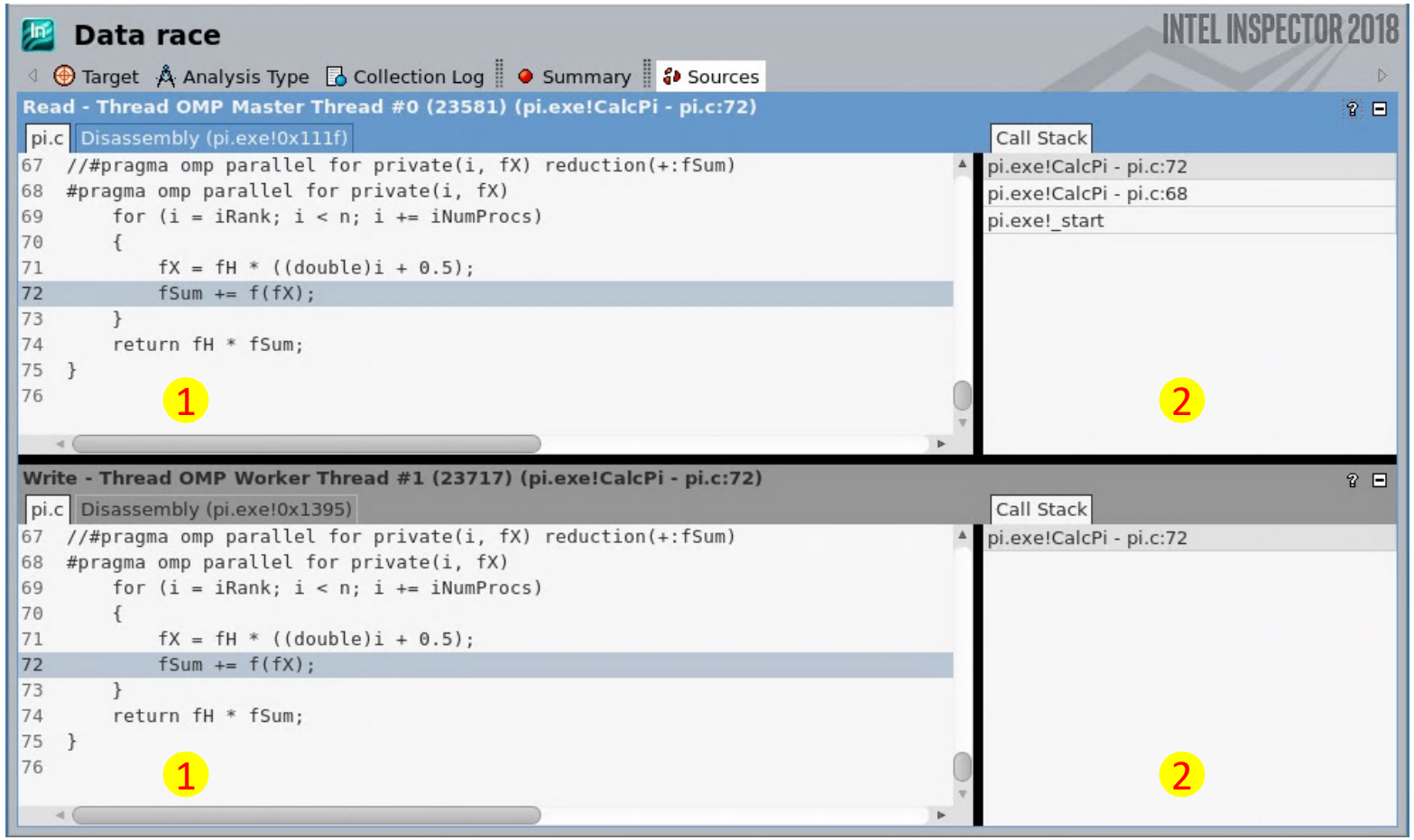
Inspector XE: results / 1

1. detected problems
2. filters
3. code location
4. Timeline



Inspector XE: results / 2

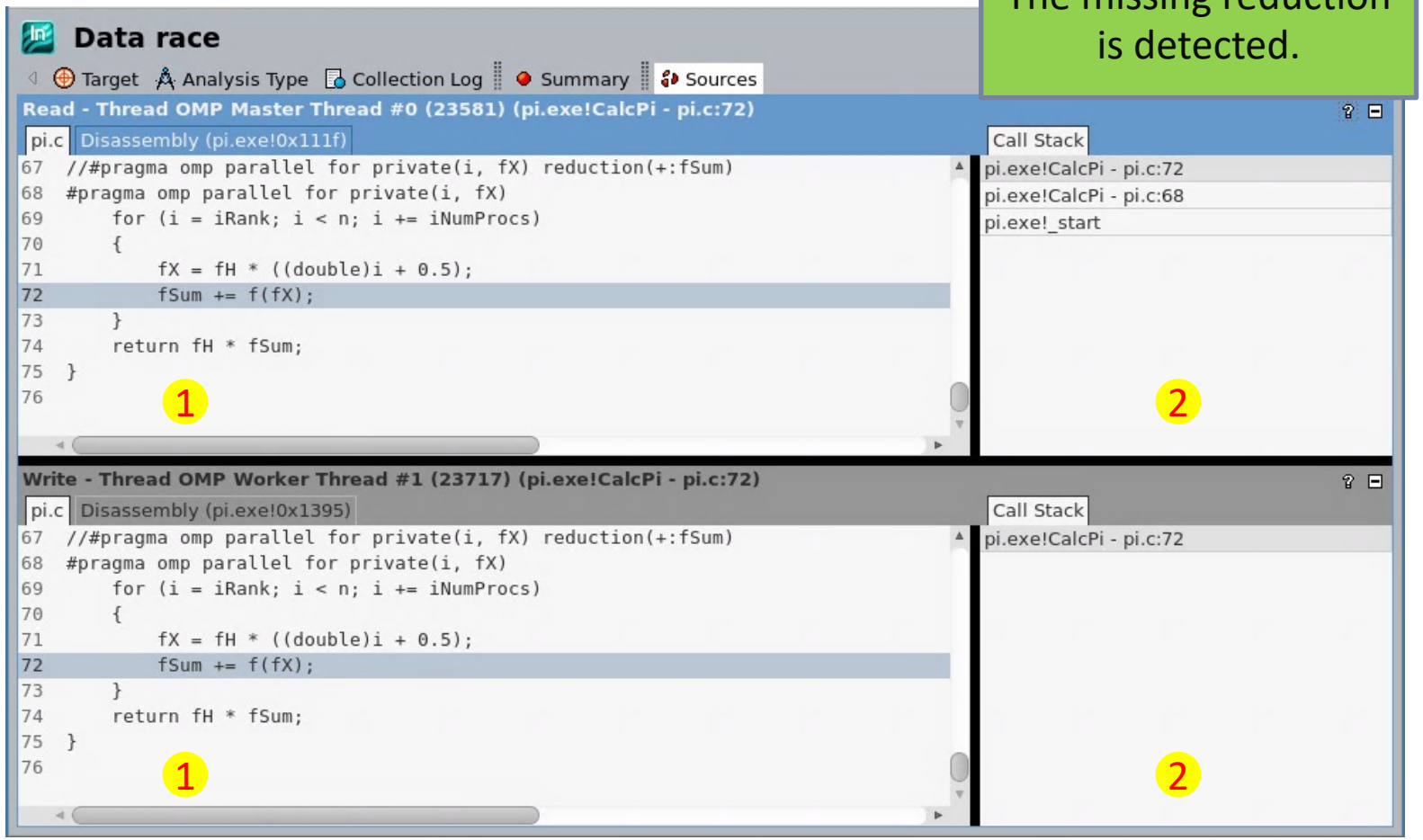
1. Source Code producing the issue – double click opens an editor
2. Corresponding Call Stack



Inspector XE: results / 3

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

The missing reduction is detected.

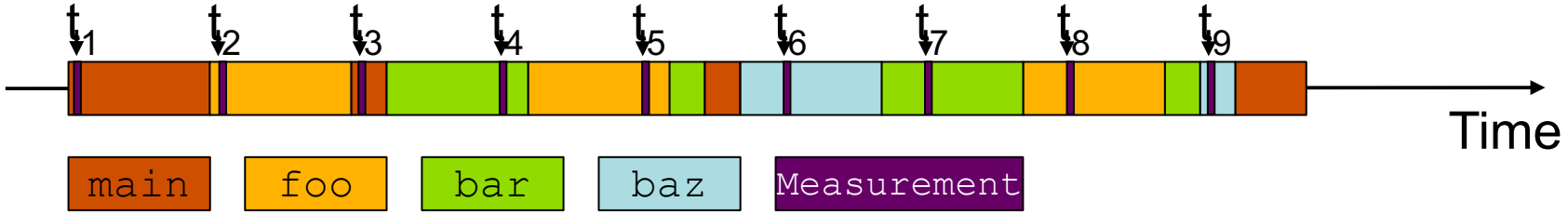


Performance Tools

Sampling vs. Instrumentation

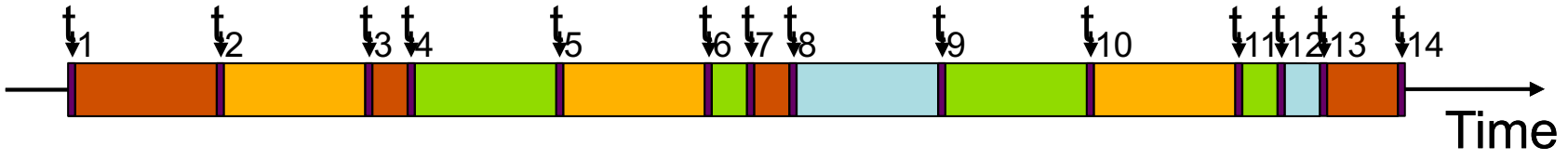
Sampling

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



Instrumentation

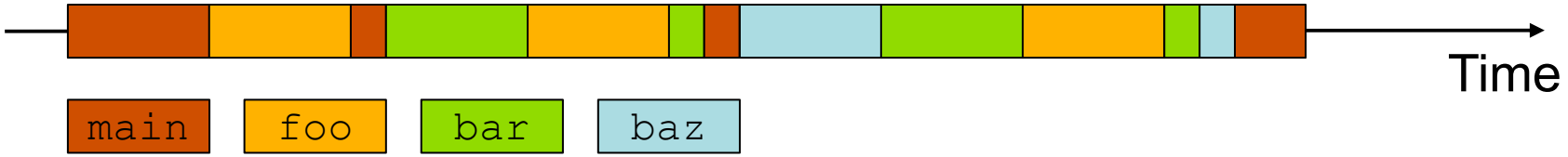
- Every event of interest is captured directly
- More detailed and *exact* information
- Typically: recompile for instrumentation



Tracing vs. Profiling

Trace

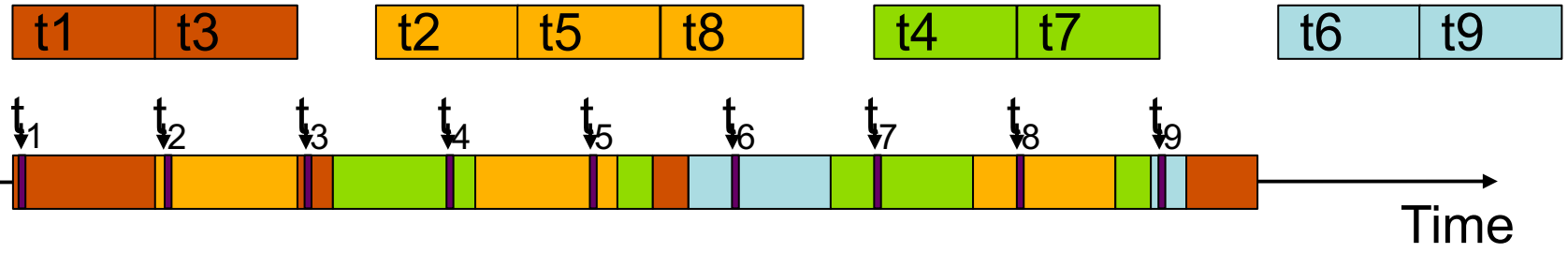
- Chronologically ordered sequence of event records



Profile from instrumentation

- Aggregated information

Profile from sampling

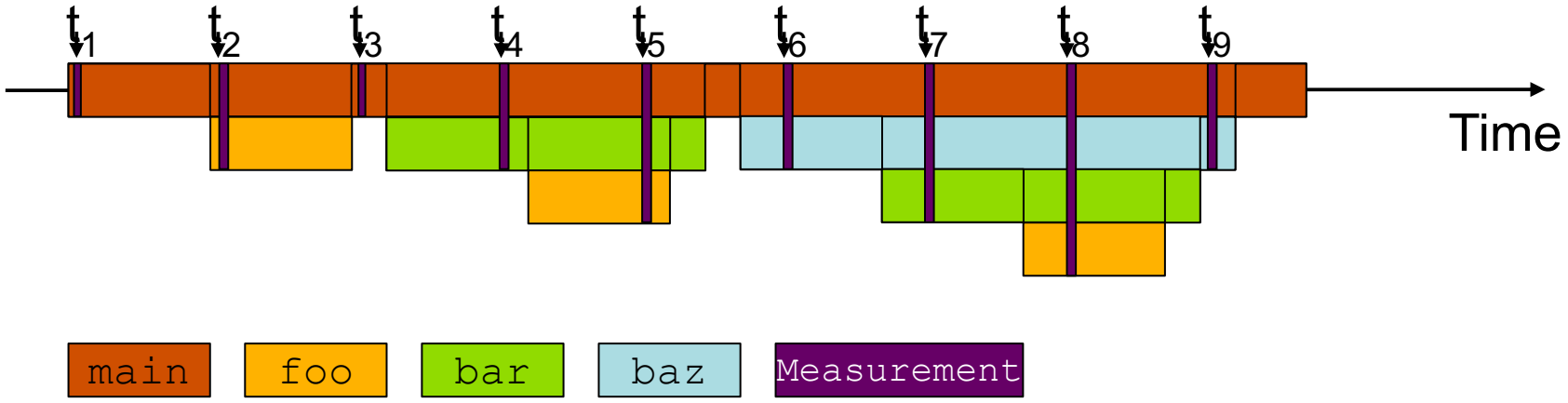


OMPT support for sampling

- 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

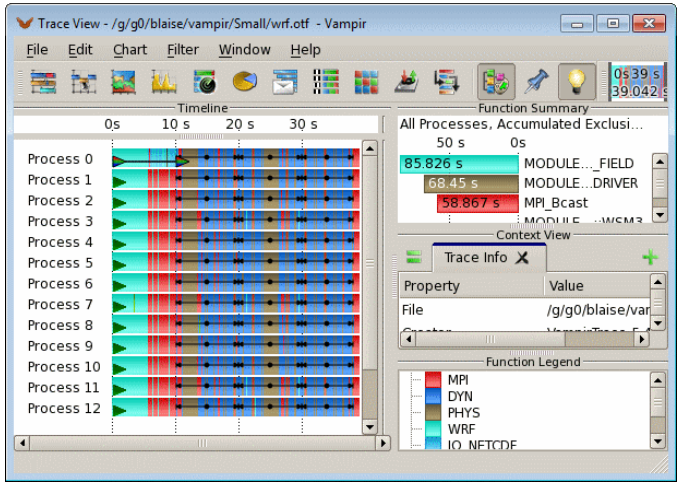
Data processing

- Scalasca : trace-based analysis

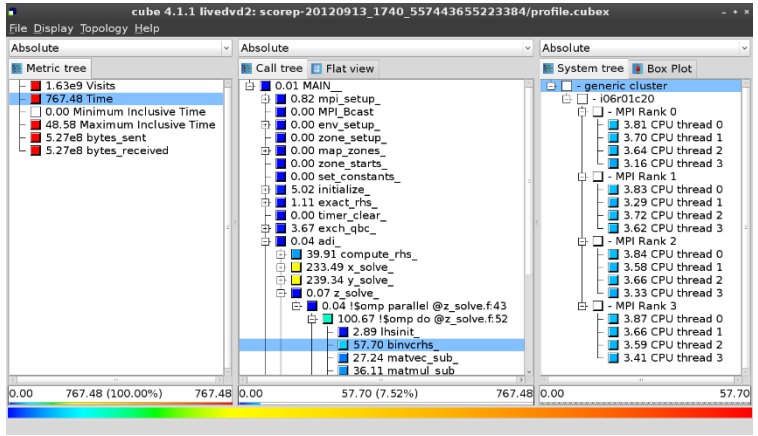
Data presentation

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

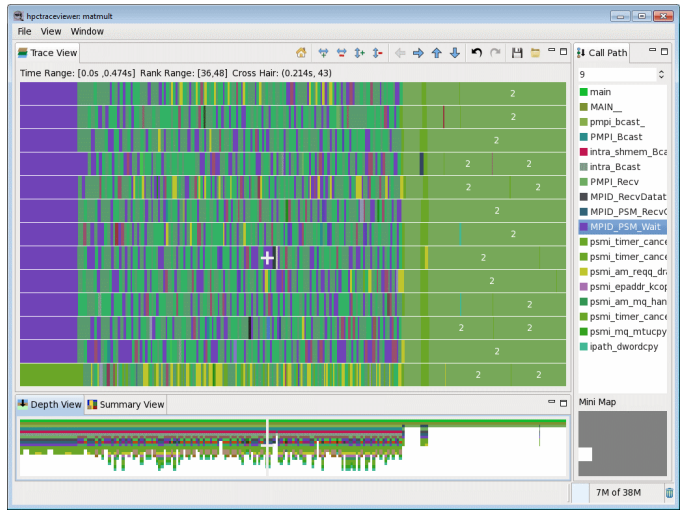
Performance tools GUI



VAMPIR



cube scalasca

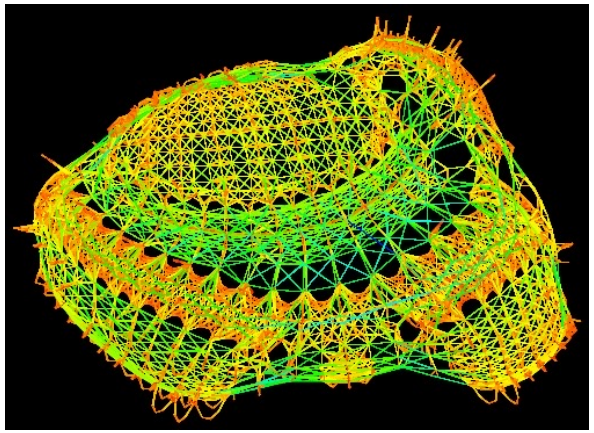


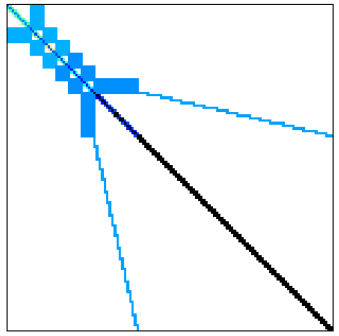
HPC Toolkit

Case Study: CG

Case Study: CG

- Sparse Linear Algebra
  - Sparse Linear Equation Systems occur in many scientific disciplines.
  - Sparse matrix-vector multiplications (SpMxV) are the dominant part in many iterative solvers (like the CG) for such systems.
  - number of non-zeros << n*n





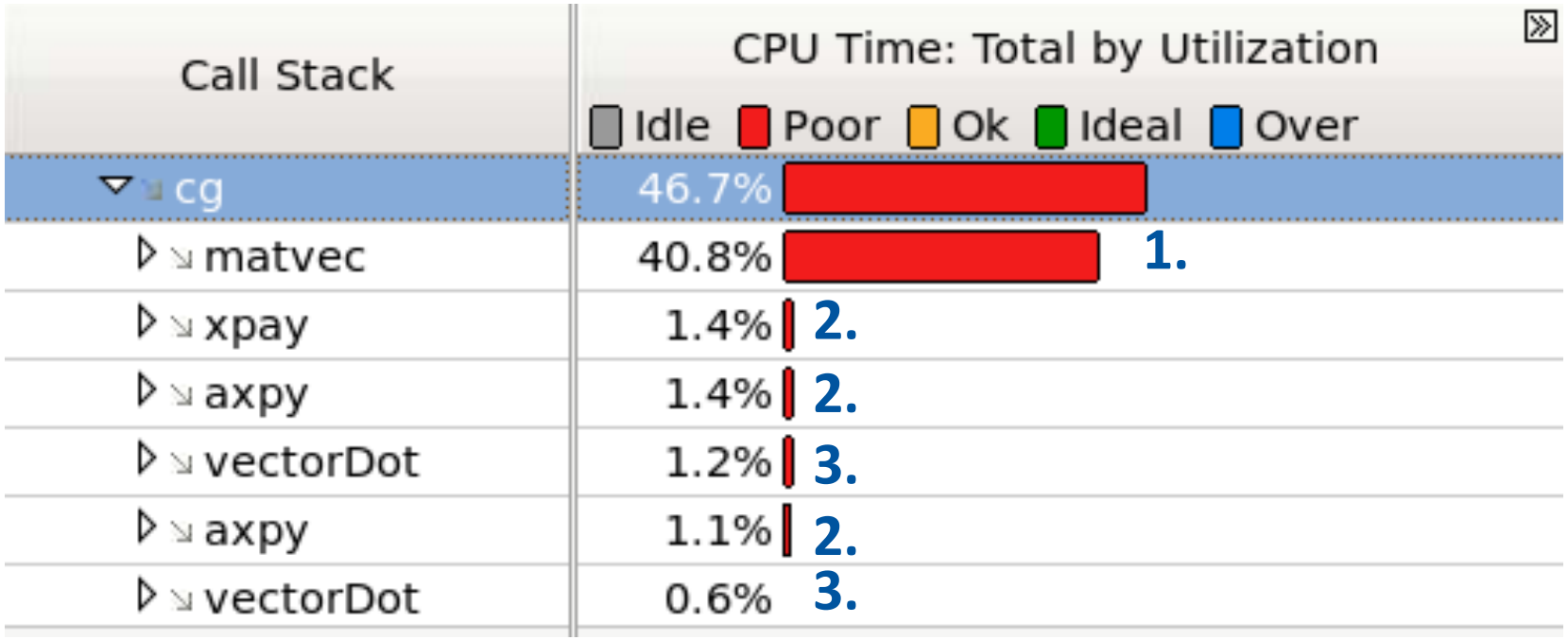
Beijing Botanical Garden

Upper Right: Orginal Building
Lower Right: Model
Lower Left: Matrix

(Source: Beijing Botanical Garden and University of Florida, Sparse Matrix Collection)

Case Study CG: Step 1

Hotspot analysis of the serial code:



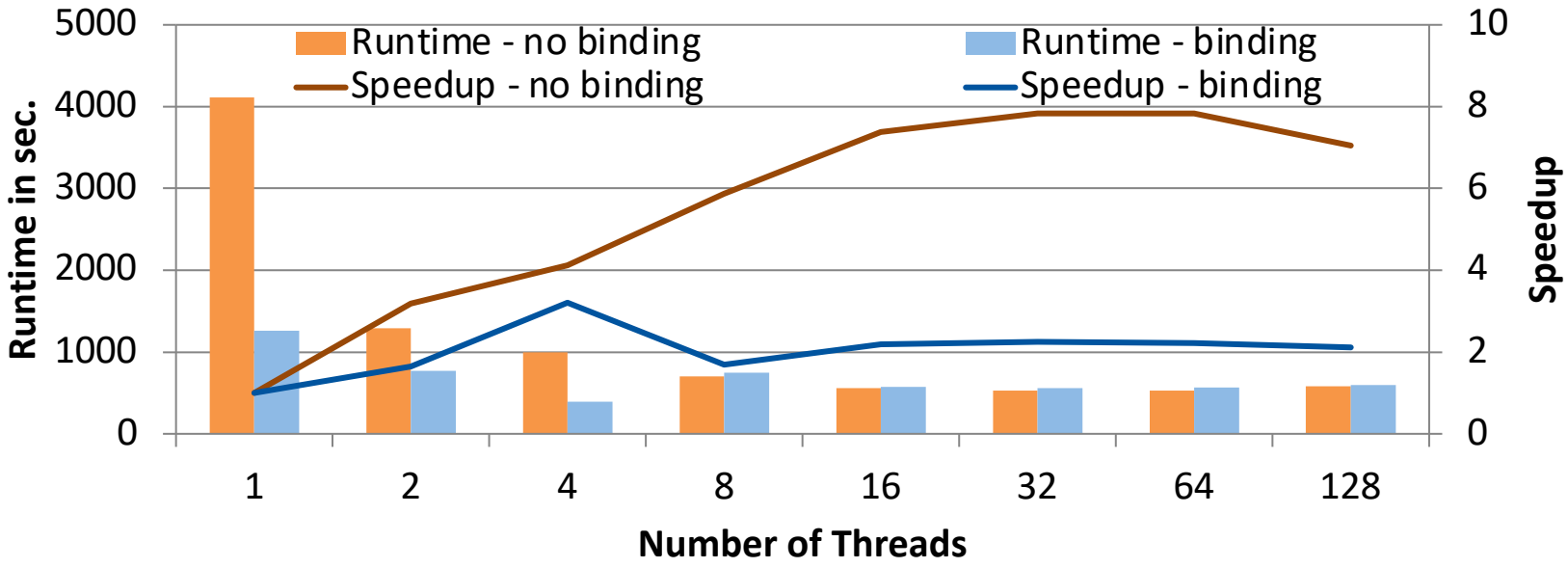
Hotspots are:

1. matrix-vector multiplication
2. scaled vector additions
3. dot product

Case Study CG: Step 1

Tuning:

- parallelize all hotspots with a `parallel for` construct
- use a reduction for the dot-product
- activate thread binding



Case Study CG: Step 2

- Hotspot analysis of naive parallel version:

Event Name
MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT
MEM_UNCORE_RETIRED.REMOTE_DRAM

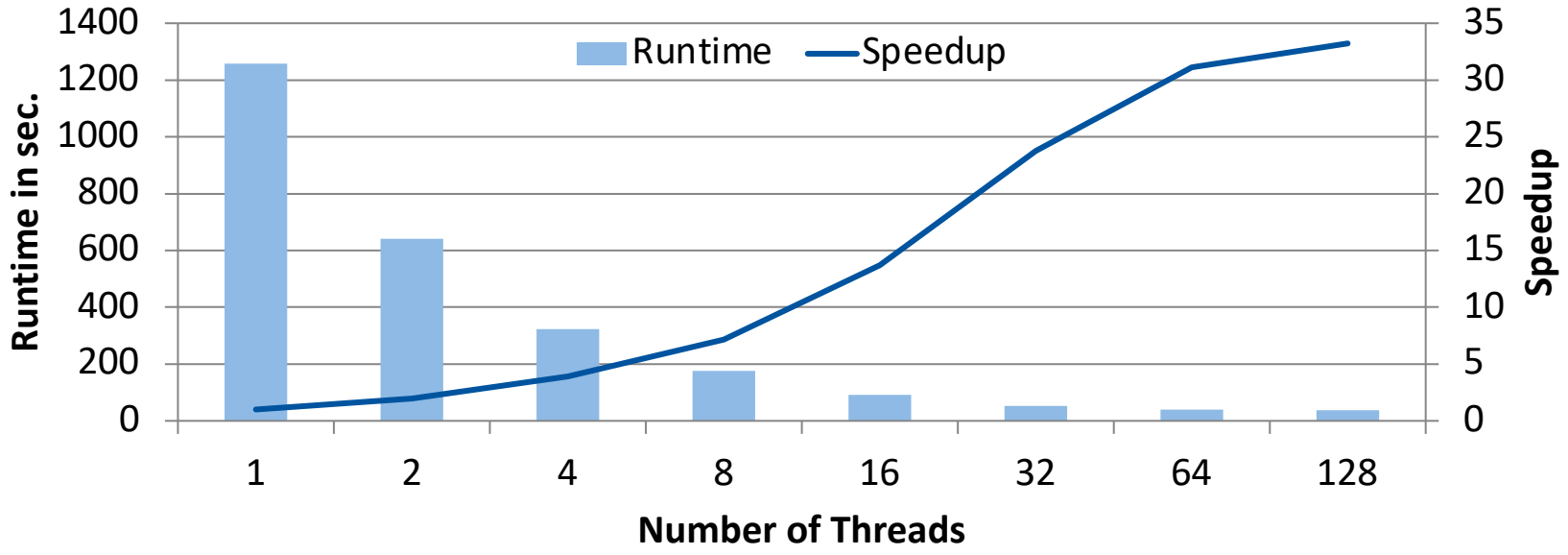
- Measurements done on a 2-socket system, because hardware counters were only available there
- A lot of remote accesses occur in nearly all places.

	MEM_UNCORE_RETIRED.LOCAL_...	MEM_UNCORE_RETIRED.REMOTE...
`void matvec(const int n, const int`		
`  int i,j;`		
`#pragma omp parallel for private(j)`	20,000	0
`  for(i=0; i<n; i++){`	0	0
`  y[i]=0;`	0	0
`  for(j=ptr[i]; j<ptr[i+1]; j`	6,740,000	3,720,000
`  y[i]+=value[j]*x[index[`	17,580,000	6,680,000
`  }`		
`  }`		

Case Study CG: Step 2

Tuning:

- Initialize the data in parallel
- Add parallel for constructs to all initialization loops



- Scalability improved a lot by this tuning on the large machine.

Case Study CG: Step 3

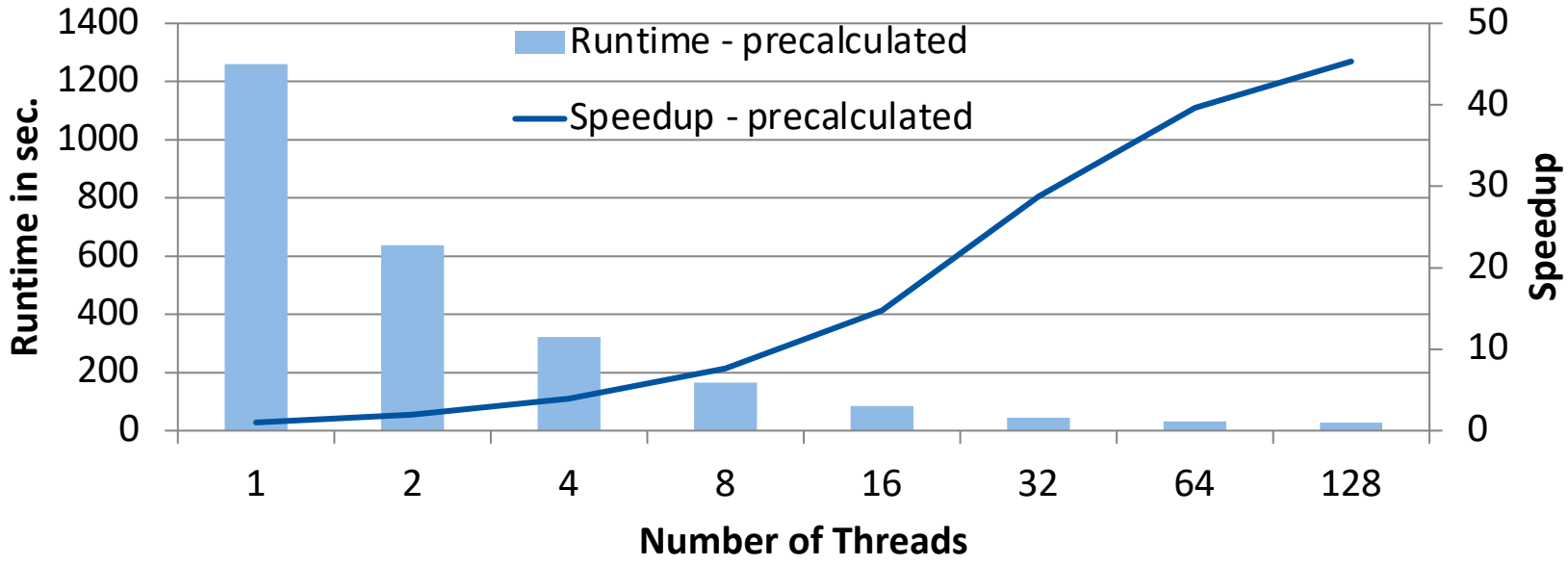
- Analyzing load imbalance in the concurrency view:

So.. Line	Source	CPU Time: Total by... Idle Poor Ok Ic	Ove... and...
49	void matvec(const int n, const int nnz,		
50	int i,j;		
51	#pragma omp parallel for private(j)	22.462s	10.612s
52	for(i=0; i<n; i++){	0.050s	0s
53	y[i]=0;	0.060s	0s
54	for(j=ptr[i]; j<ptr[i+1]; j++){	1.741s	0s
55	y[i]+=value[j]*x[index[j]];	9.998s	0s

- 10 seconds out of ~35 seconds are overhead time
- other parallel regions which are called the same amount of time only produce 1 second of overhead

Case Study CG: Step 3

- Tuning:
  - pre-calculate a schedule for the matrix-vector multiplication, so that the non-zeros are distributed evenly instead of the rows



Summary

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.

Programming OpenMP

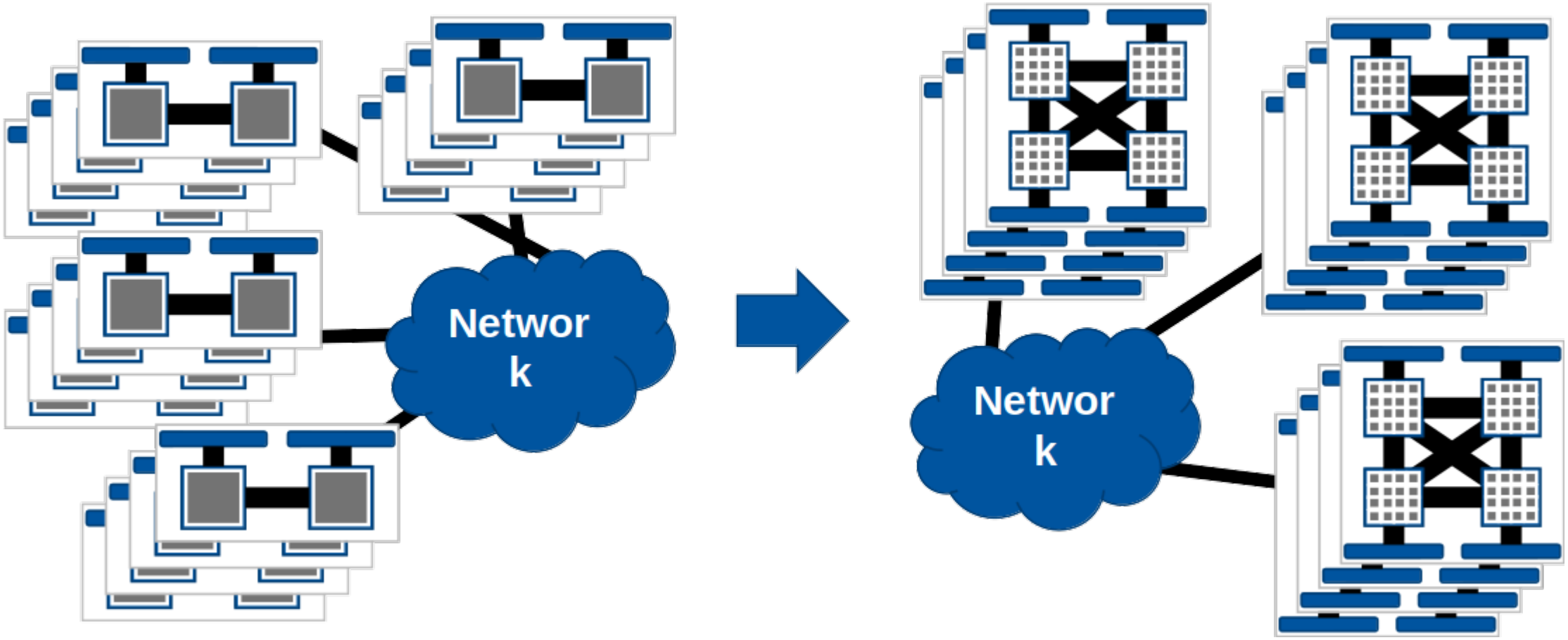
OpenMP and MPI

Christian Terboven
Michael Klemm

Motivation

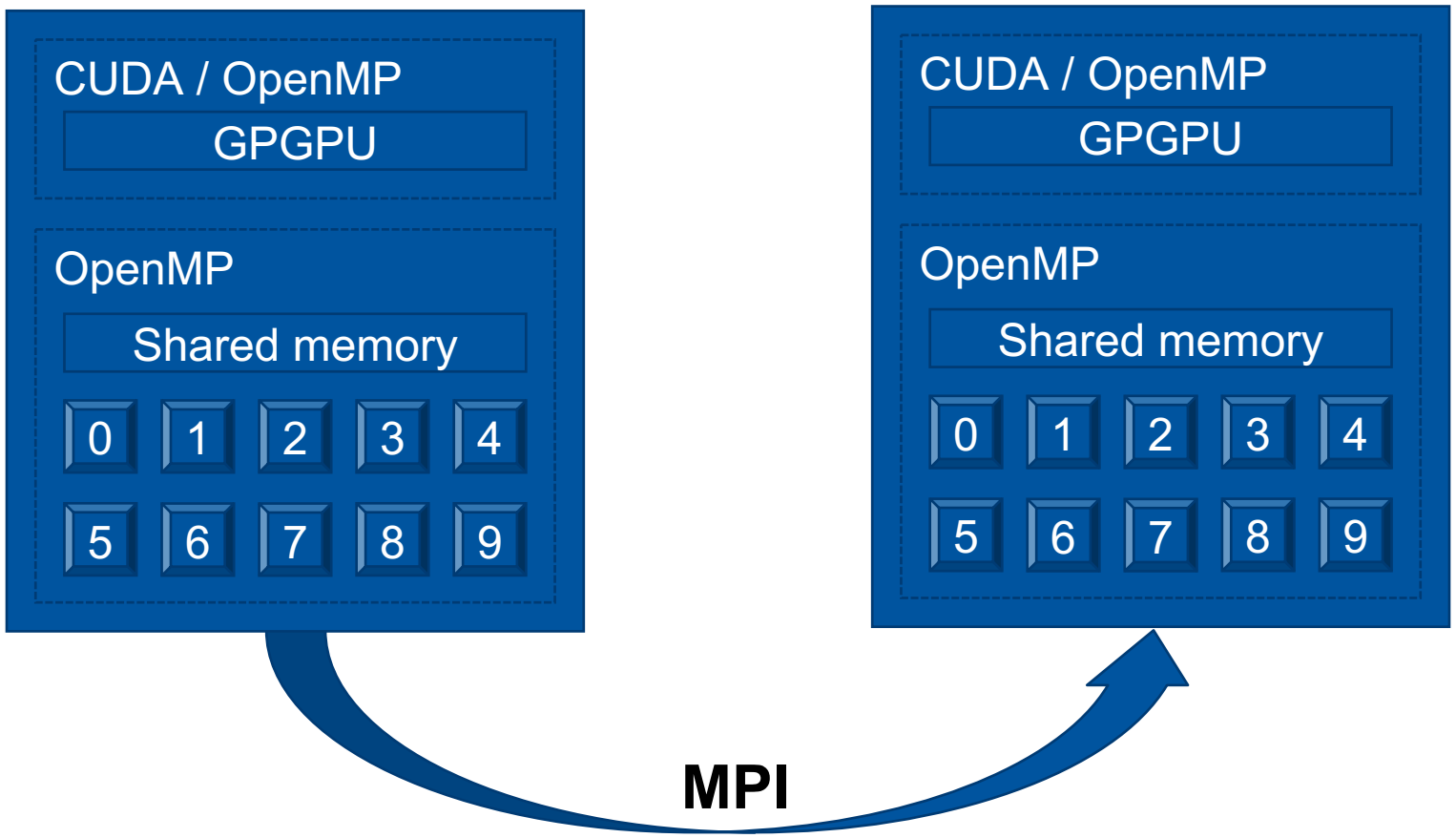
Motivation for hybrid programming

- Increasing number of cores per node



Hybrid programming

- (Hierarchical) mixing of different programming paradigms



MPI and OpenMP

MPI – threads interaction

- MPI needs special initialization in a threaded environment
  - Use `MPI_Init_thread` to communicate thread support level

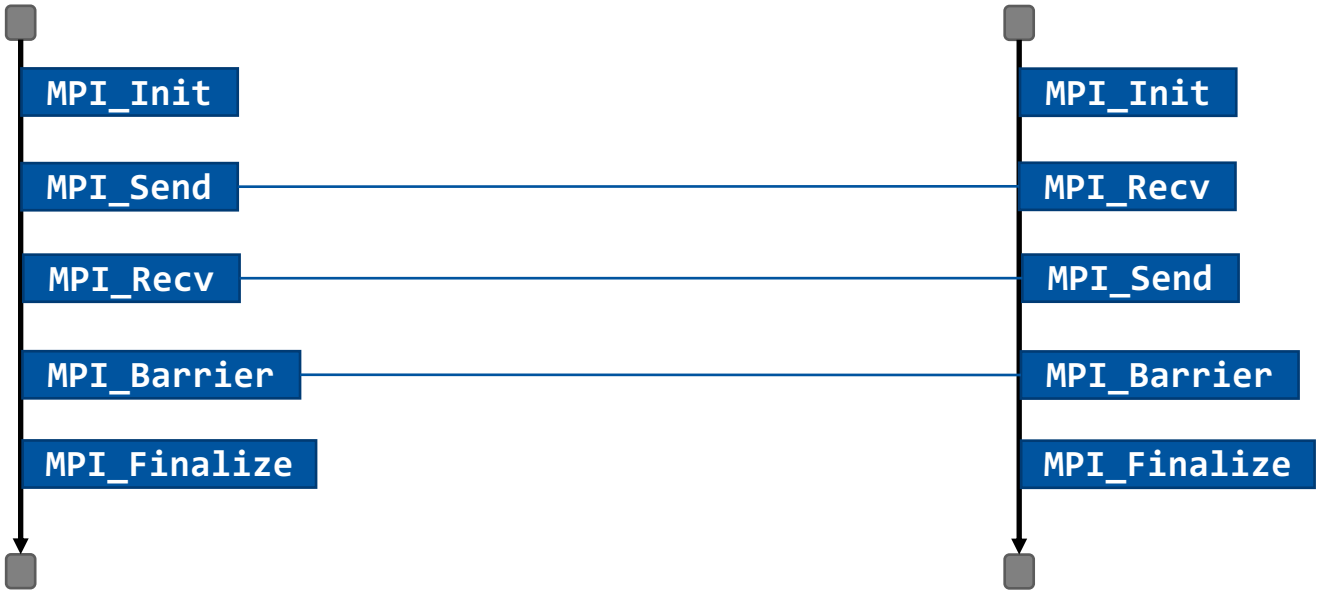
- Four levels of threading support

Level identifier	Description
`MPI_THREAD_SINGLE`	Only one thread may execute
`MPI_THREAD_FUNNELED`	Only the main thread may make MPI calls
`MPI_THREAD_SERIALIZED`	Any one thread may make MPI calls at a time
`MPI_THREAD_MULTIPLE`	Multiple threads may call MPI concurrently with no restrictions

- `MPI_THREAD_MULTIPLE` may incur significant overhead inside an MPI implementation

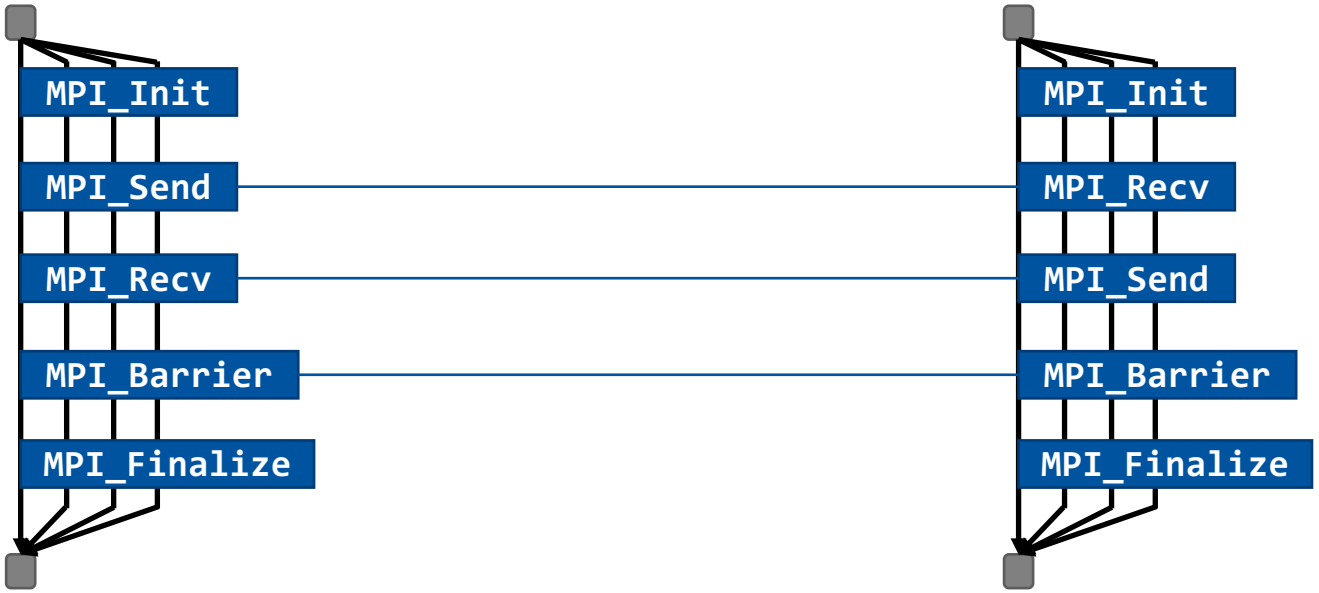
MPI – Threading support levels

- MPI_THREAD_SINGLE
  - Only one thread per MPI rank



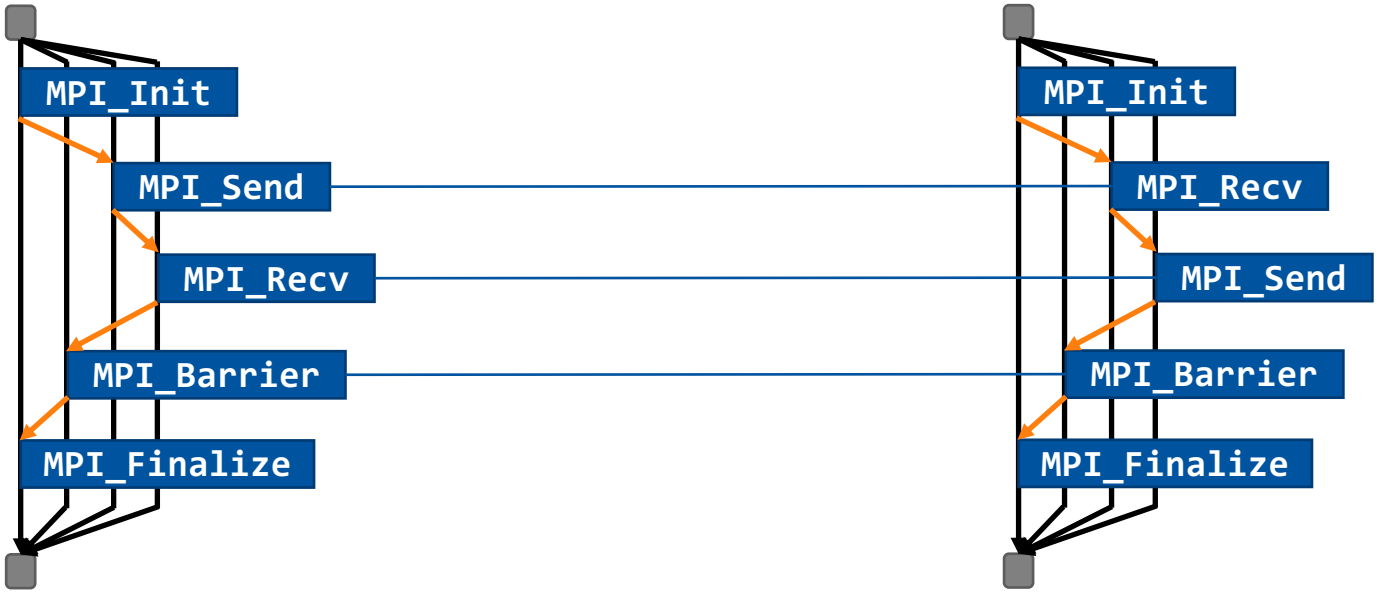
MPI – Threading support levels

- MPI_THREAD_FUNNELED
  - Only one thread communicates



MPI – Threading support levels

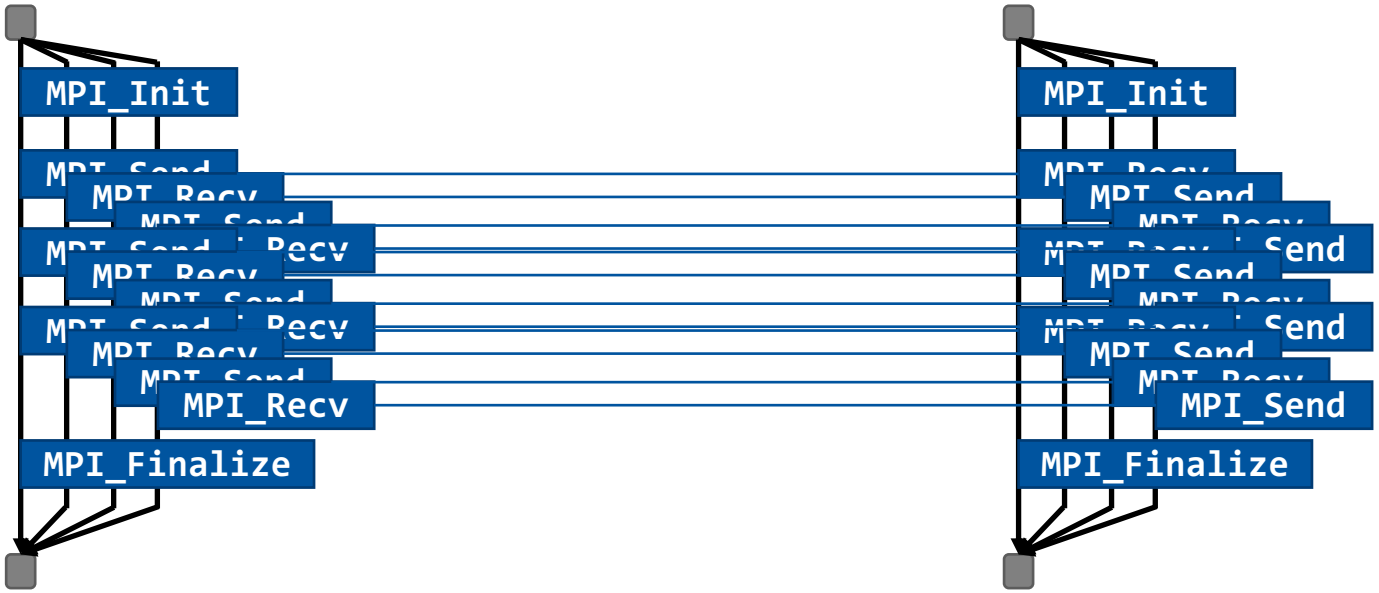
- MPI_THREAD_SERIALIZED
  - Only one thread communicates at a time



MPI – Threading support levels

- MPI_THREAD_MULTIPLE
  - All threads communicate concurrently without synchronizatio

MPI Communication
Thread Synchronization

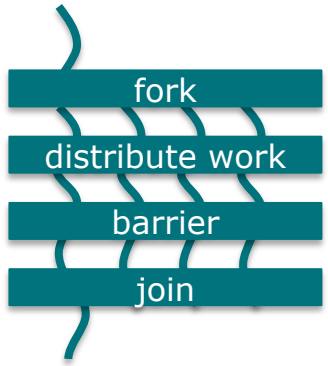


OpenMP Parallel Loops

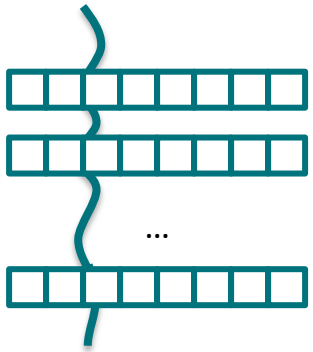
`loop` Construct

- Existing loop constructs are tightly bound to execution model:

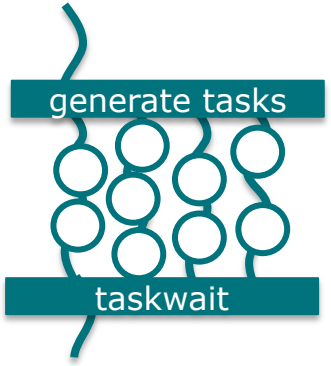
```
#pragma omp parallel for
for (i=0; i<N;++i) {…}
```




```
#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

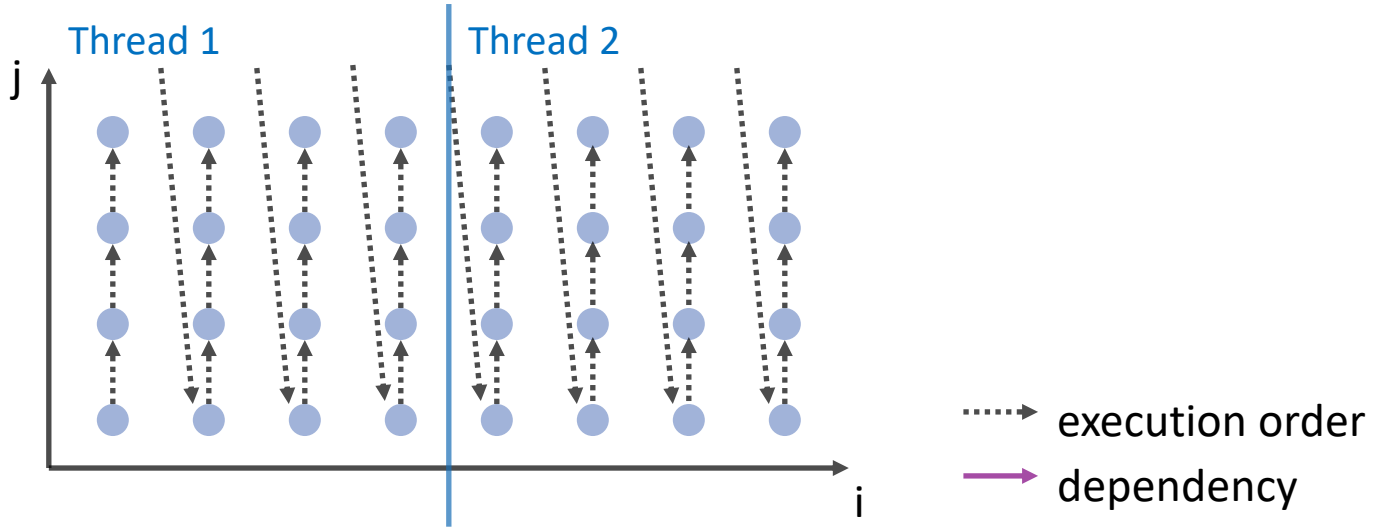
- Loop-carried dependency:
  - → Loop iterations depend on each other
  - → Source of dependency must scheduled before sink of the dependency

- 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!)

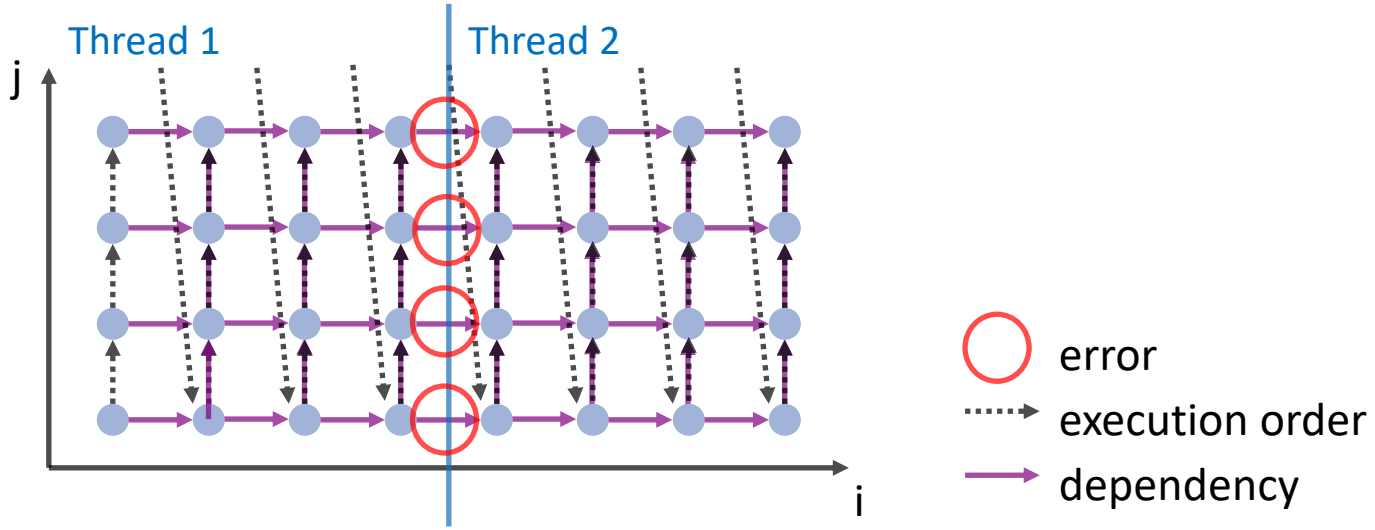
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i][j],
                    b[i][j], a[i][j]);
    }
}
```



Non-parallelizable Loops

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

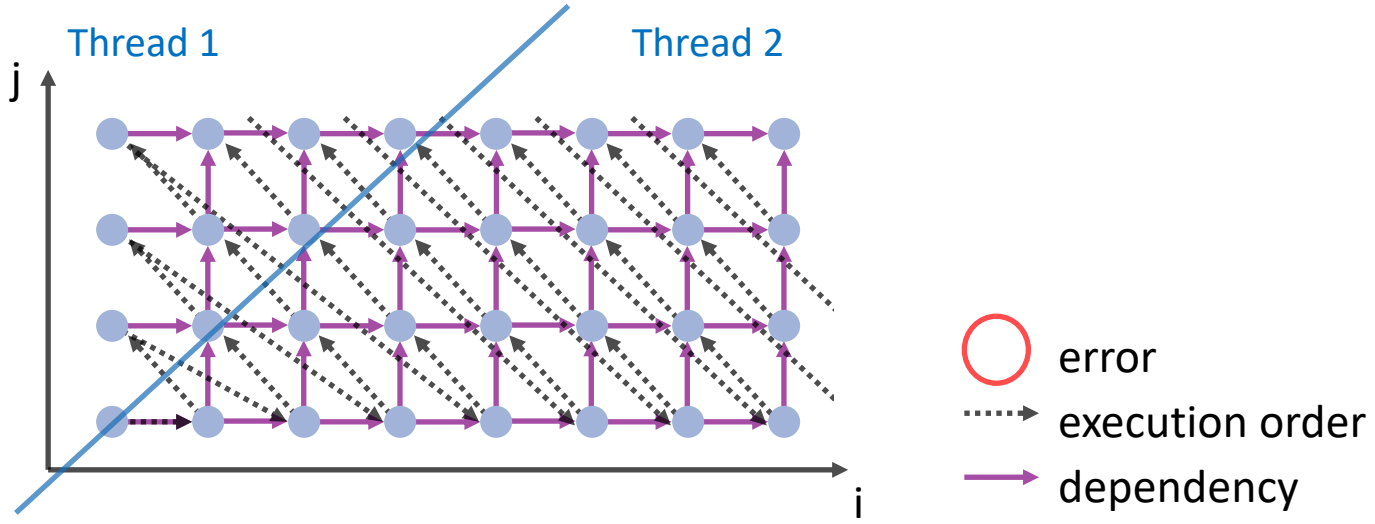
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
}
```



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)
  }
}
```

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 [clause[[,] clause],…]
for-loops
```

and

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

where clause is one of the following:

`doaccross(source:vector)`, vector can be `omp_cur_iteration`
`doaccross(sink:vector)`

- Syntax (Fortran):

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

Example

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

```
#pragma omp parallel for ordered
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered doacross(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 doacross(source:omp_cur_iteration)
}
```

Example: 3D Gauss-Seidel

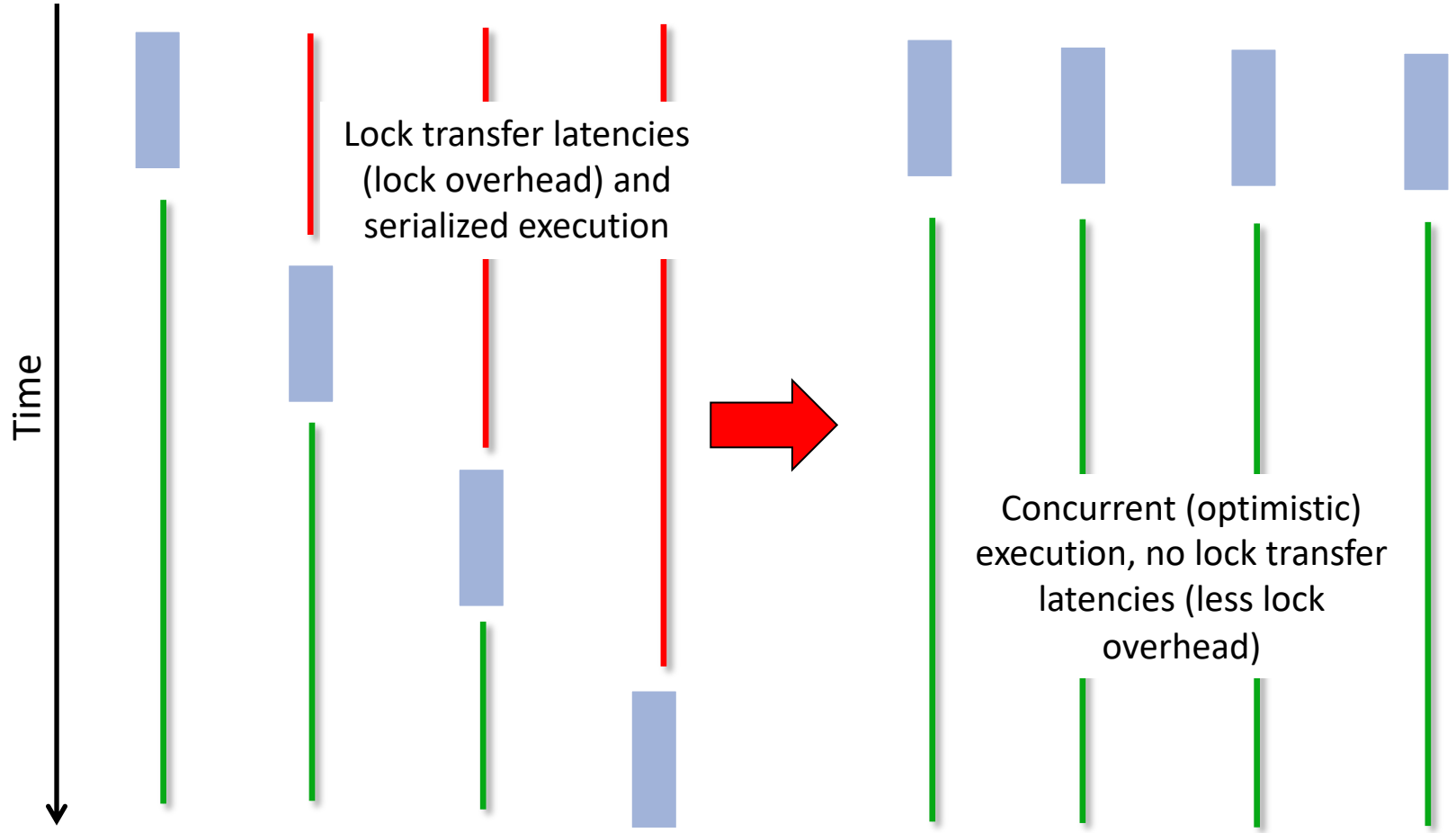
```
#pragma omp for ordered private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)   {
#pragma omp ordered doacross(sink: i-1,j-1) doacross(sink: i-1,j) \
                    doacross(sink: i-1,j+1) doacross(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + 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)
  }
}
```

Advanced Locking

Motivation

- Hardware supports new concepts for locks
  - → Intel® Transactional Synchronization Extensions
  - → Transactional memory in BlueGene*/Q
  - → Arm* Transactional Memory Extension
- Coarse-grained control does not help applications that have mixed locking requirements
  - → Some locks may be highly contended
  - → Some locks may be used to protect system calls (e.g., IO)
  - → Some locks may be just there for safety, but are almost never conflicting (e.g., hash map)
- Programmers need the ability to choose locks on a per-use basis

Lock Elision



Two new API Routines

- `omp_init_lock(omp_lock_t *lock)`
- `omp_init_lock_with_hint(omp_lock_t *lock, omp_sync_hint_t hint)`
- `omp_set_lock(omp_lock_t *lock)`
- `omp_unset_lock(omp_lock_t *lock)`
- `omp_destroy_lock(omp_lock_t *lock)`

Two new API Routines

- `omp_init_nest_lock(omp_nest_lock_t *lock)`
- `omp_init_nest_lock_with_hint(`
  `omp_nest_lock_t *lock,`
  `omp_sync_hint_t hint)`
- `omp_set_nest_lock(omp_nest_lock_t *lock)`
- `omp_unset_nest_lock(omp_nest_lock_t *lock)`
- `omp_destroy_nest_lock(omp_nest_lock_t *lock)`

Hints

- Hints are integer expressions
  - C/C++: can be combined using the `|` operator
  - Fortran: can be combined using the `+` operator

- Supported hints:
  - `omp_sync_hint_none`
  - `omp_sync_hint_uncontended`
  - `omp_sync_hint_contended`
  - `omp_sync_hint_nonspeculative`
  - `omp_sync_hint_speculative`

- OpenMP 4.5 `omp_lock_hint_*` has been deprecated

New Clause for `critical`

- Specify a hint how to implement mutual exclusion
  - → If a `hint` clause is specified, the `critical` construct must be a named construct.
  - → All `critical` constructs with the same name must have the same `hint` clause.
  - → The expression of the `hint` clause must be a compile-time constant.

- Syntax (C/C++)

```
#pragma omp critical [(name) [hint(expression)]]
structured-block
```

- Syntax (Fortran)

```
!$omp critical [(name) [hint(expression)]]
structured-block
!$omp end critical [(name)]
```

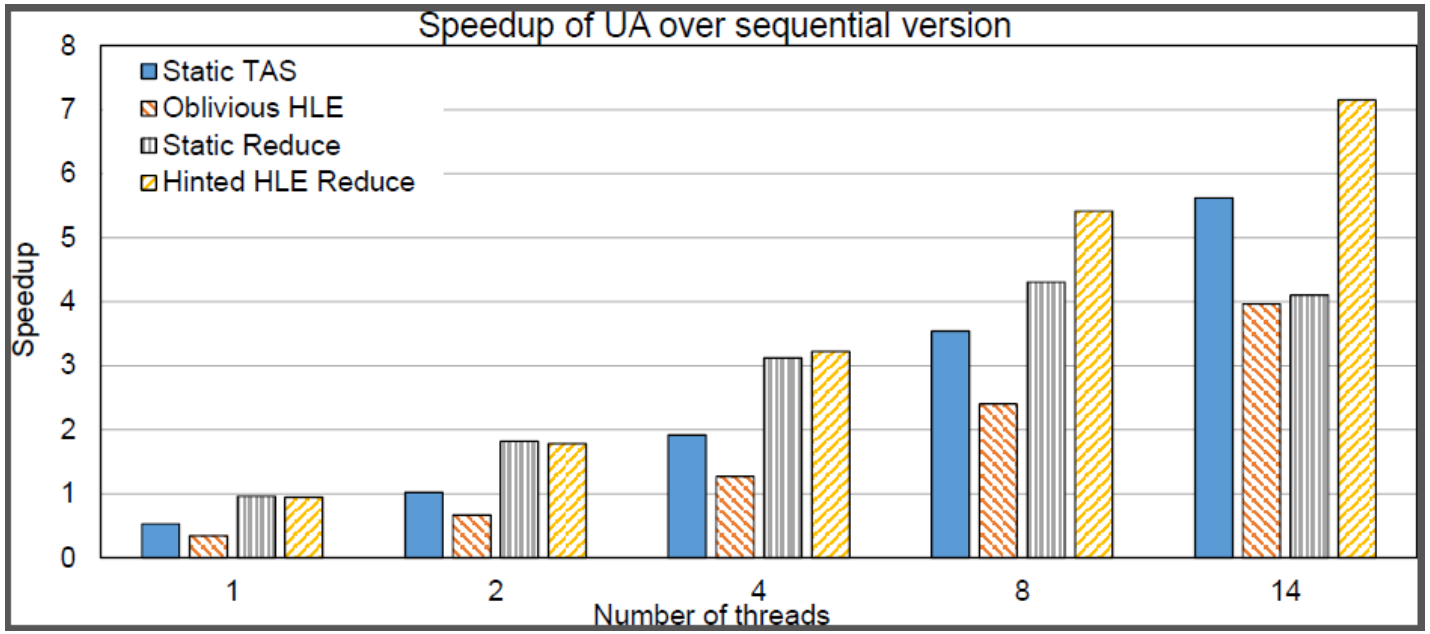
Examples

```
void example_locks() {
    omp_lock_t lock;
    omp_init_lock_with_hint(&lock, omp_sync_hint_speculative);
#pragma omp parallel
    {
        omp_set_lock(&lock);
        do_something_protected();
        omp_unset_lock(&lock);
}   }
```

```
void example_critical() {
#pragma omp parallel for
    for (int i = 0; i < upper; ++i) {
        Data d = get_some_data(i);
#pragma omp critical (HASH) hint(omp_sync_hint_speculative)
        hash.insert(d);
}   }
```

Hints May Increase Performance

- Blindly using speculative locks does not help (KMP_LOCK_KIND=…)
- Speculative locks can benefit more with growing thread counts



H. Bae, J.H. Cownie, M. Klemm, and C. Terboven. A User-guided Locking API for the OpenMP Application Program Interface. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, Using and Improving OpenMP for Devices, Tasks, and More, pages 173–186, Salvador, Brazil, September 2014. LNCS 8766.

Advanced OpenMP Tutorial

OpenMP 5.2 and Beyond

Christian Terboven

Michael Klemm

Ruud van der Pas

Bronis R. de Supinski

Future Directions

Topics

- Final Review of OpenMP 4.0, 4.5, 5.0 and 5.1
- OpenMP Organizational Overview
- Current OpenMP Language Committee Activities

Final Review of OpenMP 4.0, 4.5, 5.0 and 5.1

Ratified OpenMP 4.0 in July 2013, Ratified OpenMP 4.5 in November 2015

- OpenMP 4.0
  - → Addressed several major open issues for OpenMP
  - → Included 106 passed tickets
  - → Did not break existing code
- OpenMP 4.5
  - → Includes many refinements to 4.0 additions
  - → Included 130 passed tickets
  - → Did not break existing code unnecessarily

Overview of major 4.0 additions

- Device constructs
- SIMD constructs
- Cancellation
- Task dependences and task groups
- Thread affinity control
- User-defined reductions
- Initial support for Fortran 2003
- Support for array sections (including in C and C++)
- Sequentially consistent atomics
- Display of initial OpenMP internal control variables

Overview of major OpenMP 4.5 additions

- Many changes focused on device support
  - → Unstructured data mapping
  - → Asynchronous execution
  - → Device runtime routines: allocation, copy, etc.
  - → Clauses to support device pointers, ability to map structure elements
  - → New combined constructs
- Several other significant enhancements
  - → Support for *doacross* loops
  - → Divide loop into tasks with `taskloop` construct
  - → Hints for locks and `critical` sections
  - → Task priorities
  - → Addition of schedule modifiers: `simd`, `monotonic`, `nonmonotonic`
  - → Support for `if` clause on combined/composite constructs

Ratified OpenMP 5.0 in November 2018, Ratified OpenMP 5.1 in November 2020

- OpenMP 5.0
  - → Addressed several major open issues for OpenMP
  - → Included 293 passed tickets
  - → Did not break existing code
    - → One possible issue: `nonmonotonic` default
- OpenMP 5.1
  - → Includes many refinements to 5.0 additions
  - → Included 254 passed GitHub issues
  - → Did not break (most?) existing code
    - → Deprecated several keywords and symbols

Major new features in OpenMP 5.0

- Significant extensions to improve usability
  - → OpenMP contexts, `metadirective` and `declare variant`
  - → Addition of `requires` directive, including support for unified shared memory
  - → Memory allocators and support for deep memory hierarchies
  - → Descriptive `loop` construct
  - → Ability to quiesce OpenMP threads
  - → Support to print/inspect affinity state
  - → Release/acquire semantics added to memory model
  - → Support for C/C++ array shaping
- First (OMPT) and third (OMPD) party tool support

Major new features in OpenMP 5.0

- Some significant extensions to existing functionality
  - → Verbosity reducing changes such as implicit `declare target` directives
  - → User defined mappers provide deep copy support for map clauses
  - → Support for reverse offload
  - → Support for task reductions , including on `taskloop` construct, task affinity, new dependence types, depend objects and detachable tasks
  - → Allows `teams` construct outside of `target` construct (i.e., on host)
  - → Supports collapse of non-rectangular loops
  - → Scan extension of reductions
- Major advances for base language normative references
  - → Completed support for Fortran 2003
  - → Added Fortran 2008, C11, C++11, C++14 and C++17

OpenMP 5.0 clarifications and enhancements

- Supports collapse of imperfectly nested loops
- Supports `!=` on C/C++ loops
- Adds `conditional` modifier to `lastprivate`
- Support use of any C/C++ *lvalue* in `depend` clauses
- Permits `declare target` on C++ classes with virtual members
- Clarification of `declare target` C++ initializations
- Adds `task` modifier on many `reduction` clauses
- Adds `depend` clause to `taskwait` construct

OpenMP 5.1 refines existing functionality

- Adds full support for C11, C++11, C++14, C++17, C++20 and Fortran 2008 and partial support for Fortran 2018
- Extends directive syntax to C++ attribute specifiers
- The `scope` construct supports reductions within parallel regions
  - → Christian discussed this enhancement in another session
- Extends `atomic` construct to support compare-and-swap, min and max
  - → Detailed these enhancements in another session
- Adds many clauses and clause modifiers including:
  - → `nowait` to `taskwait` construct
  - → `strict` modifier to clauses on the `taskloop` construct

OpenMP 5.1 refines existing functionality

- Support for mapping (translated) function pointers
- Device-specific environment variables to control their ICVs
- `nothing` directive supports `metadirective` clarity and completeness
- Several new runtime routines, including more memory allocation flavors
- Deprecations include:
  - → The `master` affinity policy and `master` construct
  - → Cray pointers
  - → Many enum values, most related to OMPT (first-party tool interface)

Significant OpenMP 5.1 Features

OpenMP 5.1 adds some significant extensions

- The `interop` construct
  - → Improves native device support (e.g., CUDA streams)
  - → Also supports interoperability with CPU-based libraries (e.g., TBB)
- The new `dispatch` construct, improved `declare variant` directive
  - → Enable use of variants with device-specific arguments
  - → Elision of “unrecognized” code

OpenMP 5.1 adds some significant extensions

- The `assume` directive
  - → Supports optimization hints based on invariants
  - → Supports promise to limit OpenMP usage to (optimizable) subsets
- Loop transformation directives: The `tile` and `unroll` directives
  - → Control use of traditional sequential optimizations
  - → Ensure that they are applied when, where appropriate relative to parallelization

New Error Directive

The `error` directive supports user-defined warnings and errors

- Use `error` directive to interact with the compiler

```
#pragma omp error [at(compilation|execution)] [severity(fatal|warning)] \
                  [message(msg-string)]
  structured-block
```

- Compiler displays `msg-string` as part of implementation-defined message
- The `at` clause determines when the effect of the directive occurs
  - → `compilation`: If encountered during compilation in a declarative context (useful along with `metadirective`) or is reachable at runtime
  - → `execution`: If the code location is encountered during execution (similar to `assert()`)
- The `severity` clause determines compiler action
  - → `warning`: Print message only (default)
  - → `fatal`: Stop compilation or execution

New Masked Construct

The `masked` construct supports filtering execution per thread

- Use `masked` construct to limit parallel execution (low cost: no data environ.)

```
#pragma omp masked [filter(integer-expression)]
  structured-block
```

- Encountering thread executes if `filter` clause matches its thread number
- Default (i.e., no clause) is equivalent to deprecated `master` construct
- Future (i.e., OpenMP 6.0) enhancements planned
  - → Define concept of thread groups, a subset of the threads in a team
  - → Extend `masked` to `filter` based on thread groups or booleans (via clause modifier)
  - → `filter` clause added to other constructs, relying on thread group concept

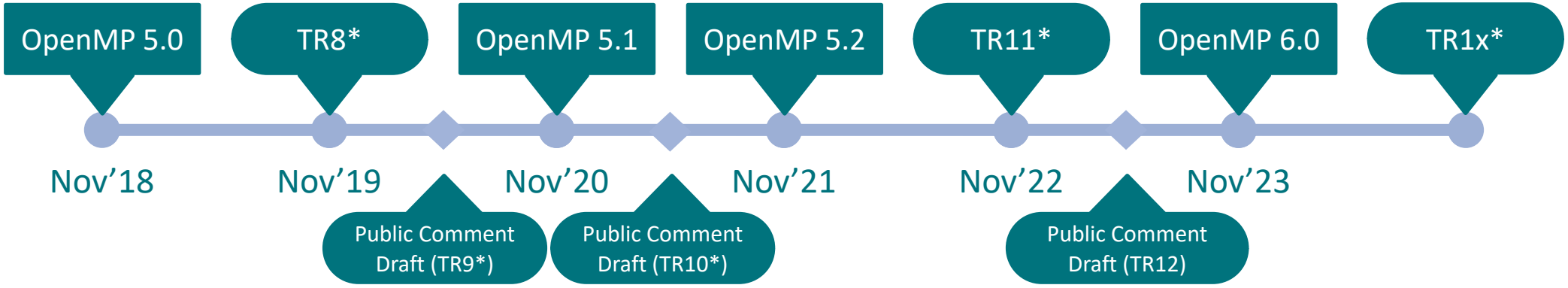
OpenMP Organizational Overview

OpenMP API Specification as a Book

- 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

OpenMP Roadmap

- OpenMP has a well-defined roadmap:
  - →5-year cadence for major releases
  - →One minor release in between
  - →OpenMP 5.2 was added as a second minor release before OpenMP version 6.0
  - →(At least) one Technical Report (TR) with feature previews in every year



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

Development Process of the Specification

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

- Release process for specifications:

User Outreach & Education

Check out openmp.org/news/events-calendar/

OpenMP Language Committee Current Activities: OpenMP 5.2 and 6.0

IEEE Proceedings article on vision for OpenMP: "The Ongoing Evolution of OpenMP"

- Broadly support on-node performant, portable parallelism
  - → Standardize syntax for commonly available (parallel) directives
  - → Consistently apply across C, C++ and Fortran
  - → To be simple yet flexible, supporting range of parallelism models
- OpenMP 5.0 fits within that vision
- OpenMP 5.1 and OpenMP 5.2 refine how OpenMP 5.0 realizes it
- OpenMP 6.0 will be a major step to further realizing it

OpenMP 5.2 was released earlier this month

- OpenMP ARB adopted on November 11, 2021
- Large portions of specification now generated from JSON-based database
  - → Section headers and directive and clause format
  - → Cross references, index entries, hyperlinks and many other document details
  - → Long-term plan will capture sufficient information in database to generate much more, including grammar, quick reference guide, and header and runtime library routine stub files
- Improves specification of OpenMP syntax
  - → Ensuring syntax of directives and clauses is well-specified and consistent
  - → Ensuring restrictions are consistent and not just implied by syntax
  - → Deprecating one-off syntax choices, many other inconsistencies (12 new deprecation entries)
  - → Makes C++ attribute syntax a first-class citizen
- Many other minor improvements
- ~125 passed issues

OpenMP 6.0 will be released in November 2023

- Removal of features that were deprecated in 5.0, 5.1 or 5.2
- Further adoption of the database-specification approach
- Dependences and affinity for the `taskloop` construct
- Task-only or free-agent threads
- Spawning tasks for other teams (event-driven parallelism and more)
- Continued improvements to device support
  - → True support for using multiple devices
  - → Extensions of deep copy support (serialize/deserialize functions)
- More support for memory affinity and complex hierarchies
- Deeper support for descriptive and prescriptive control
- Support for pipelining, other computation/data associations; data-flow?
- 161 issues already created for/deferred to 6.0

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 let us know if you would be interested