Programming OpenMP

Christian Terboven

Michael Klemm

Agenda (in total ~~5~~ webinars)

- Webinar 1: OpenMP Introduction
- Webinar 2: Tasking
- Webinar 3: Optimization for NUMA and SIMD
- Webinar 4: Introduction to Offloading with OpenMP
- Webinar 5: Advanced Offloading Topics
- **Webinar 6: Selected / Remaining Topics**
  - → Tasking: Cut-off
  - → Tasking: Affinity
  - → Hybrid Programming: Detached Tasks
  - → Hybrid Programming: MPI + OpenMP

Programming OpenMP

Cut-off strategies

Christian Terboven

Michael Klemm

Improving Tasking Performance:
Cutoff clauses and strategies

Example: Sudoku revisited

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

- (1) Search an empty field

first call contained in a
`#pragma omp parallel`
`#pragma omp single`
such that one tasks starts the execution of the algorithm

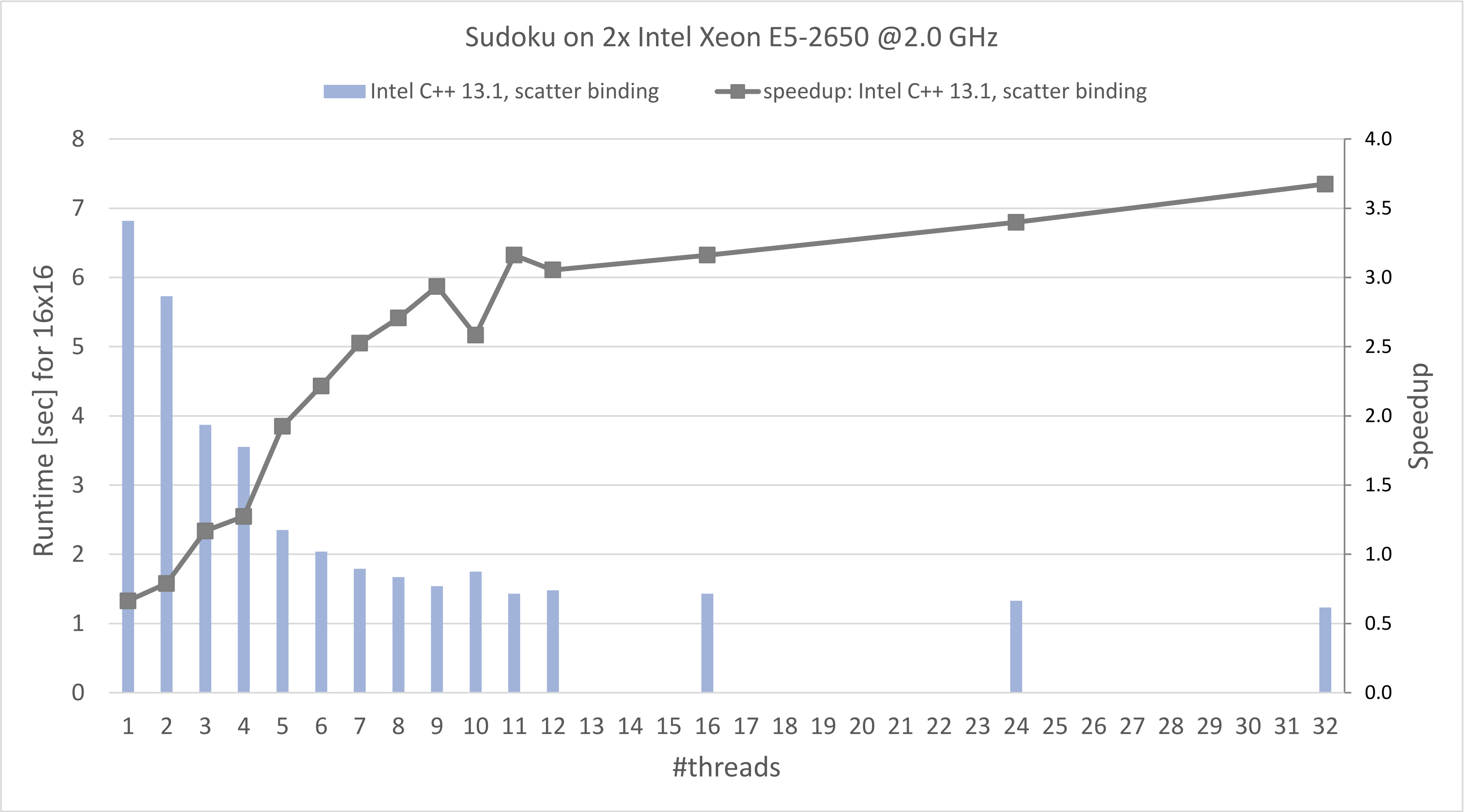
- (2) Try all numbers:
  - (2 a) Check Sudoku
    - If invalid: skip
    - If valid: Go to next field

`#pragma omp task`
needs to work on a new copy of the Sudoku board

- Wait for completion

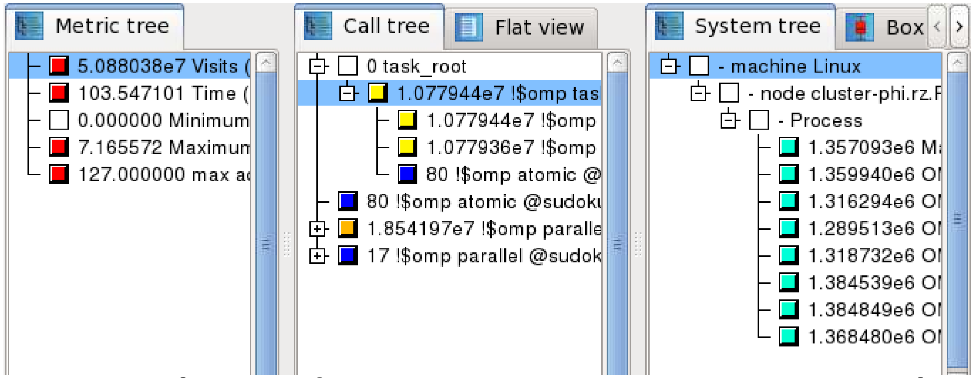
`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation

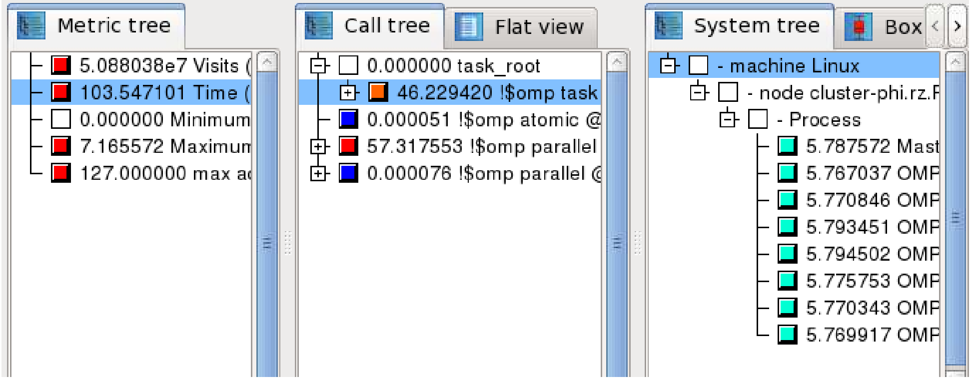


Performance Analysis

Event-based profiling provides a good overview :



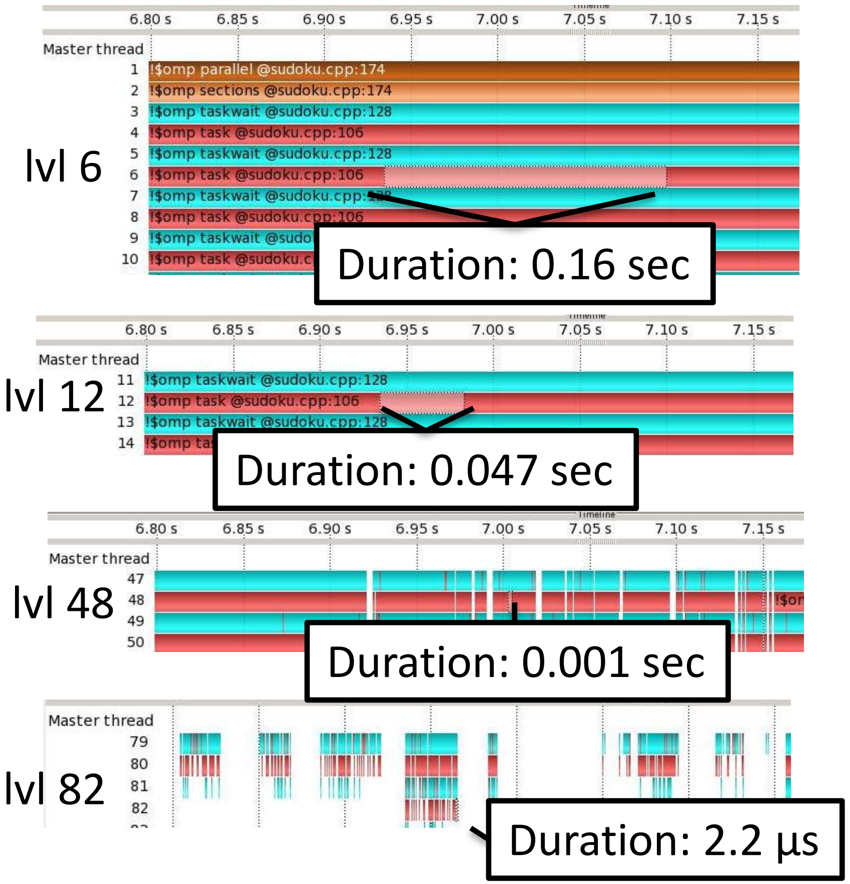
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

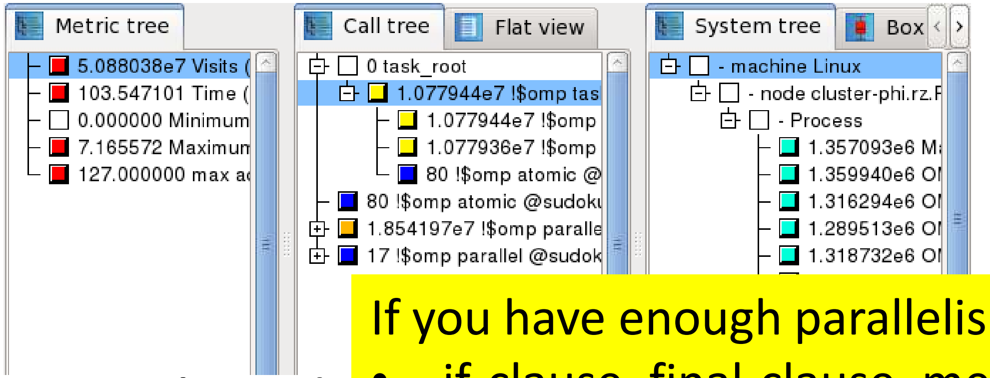
Tracing provides more details:



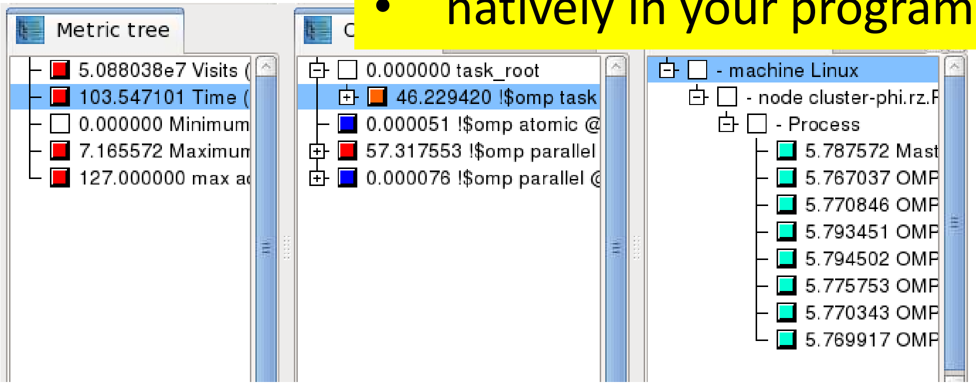
Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling provides a good overview :



Every thread i

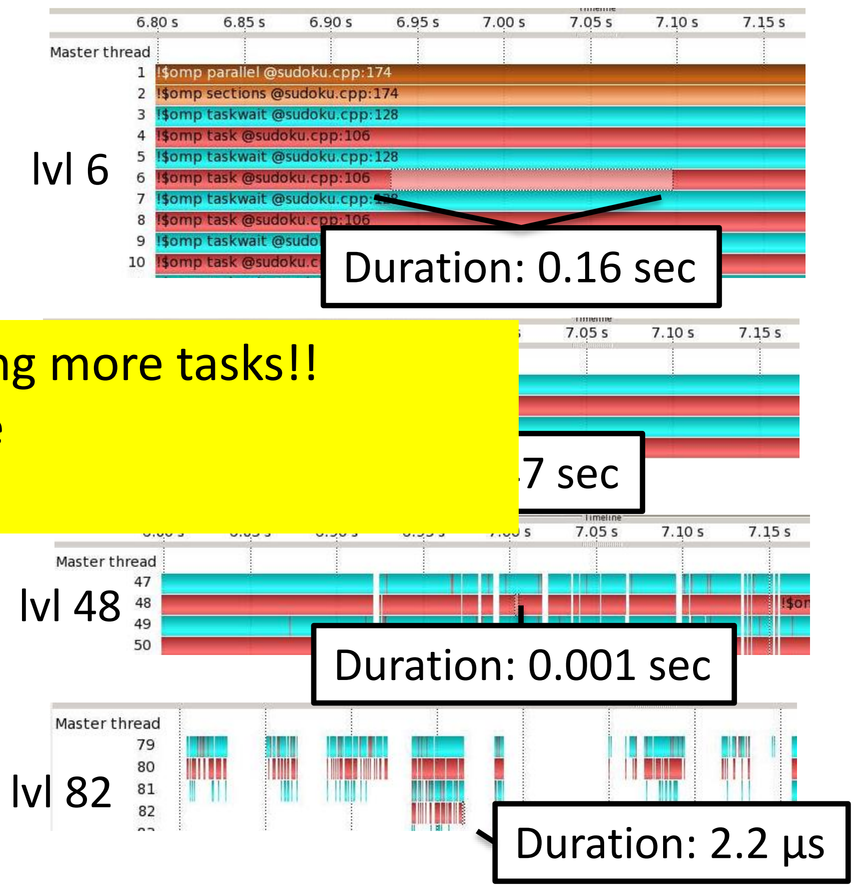


If you have enough parallelism, stop creating more tasks!!

- if-clause, final-clause, mergeable-clause
- natively in your program code

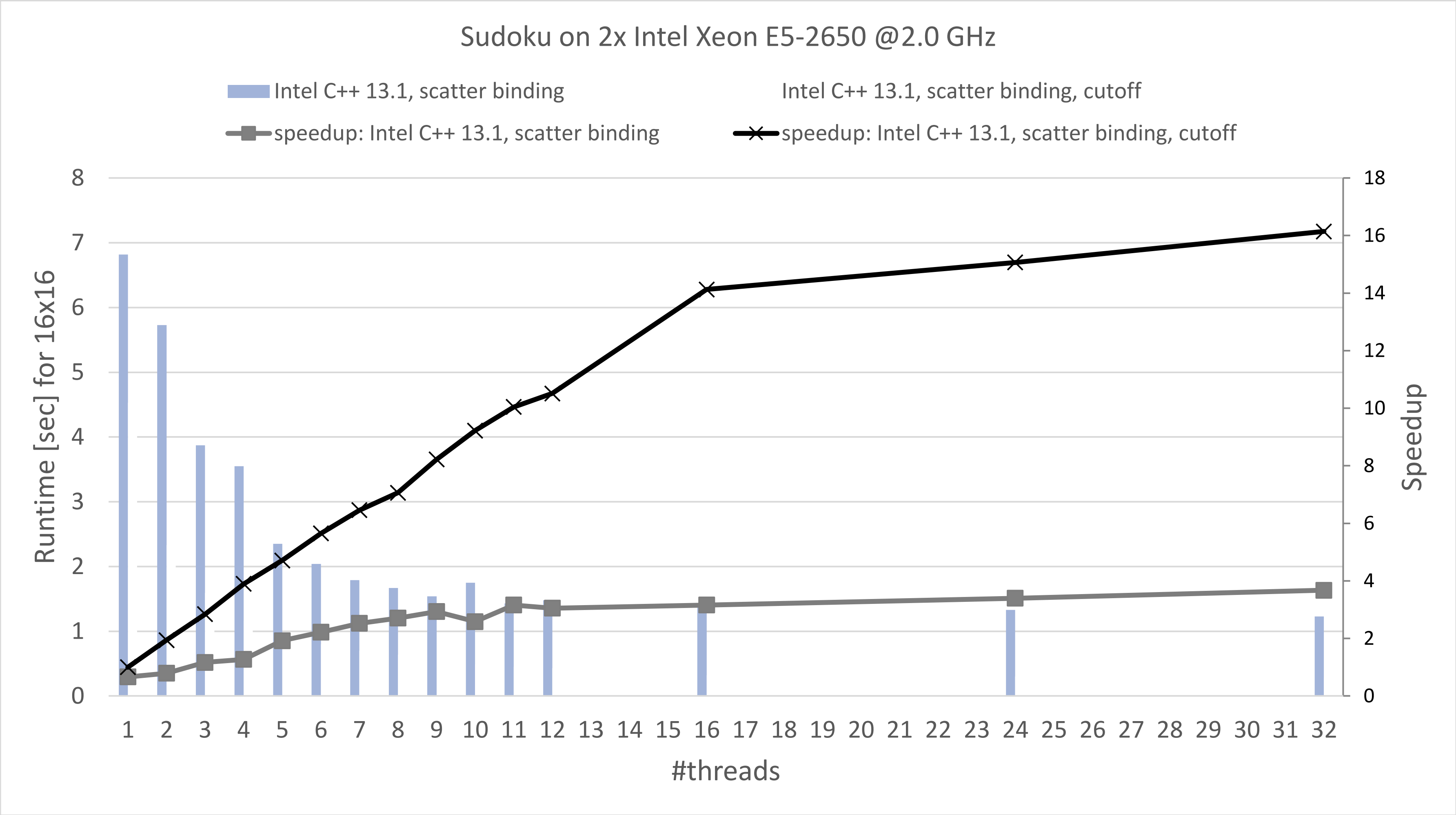
... in ~5.7 seconds.
=> average duration of a task is ~4.4 µs

Tracing provides more details:



Tasks get much smaller down the call-stack.

Performance Evaluation (with cutoff)



The `if` clause

- Rule of thumb: the `if(expression)` clause as a “switch off” mechanism
  - → Allows lightweight implementations of task creation and execution but it reduces the parallelism
- If the `expression` of the `if` clause evaluates to `false`
  - → the encountering task is suspended
  - → the new task is executed immediately (task dependences are respected!!)
  - → the encountering task resumes its execution once the new task is completed
  - → This is known as *undeferred task*

```
int foo(int x) {
  printf(“entering foo function\n”);
  int res = 0;
  #pragma omp task shared(res) if(false)
  {
        res += x;
  }
  printf(“leaving foo function\n”);
}
```

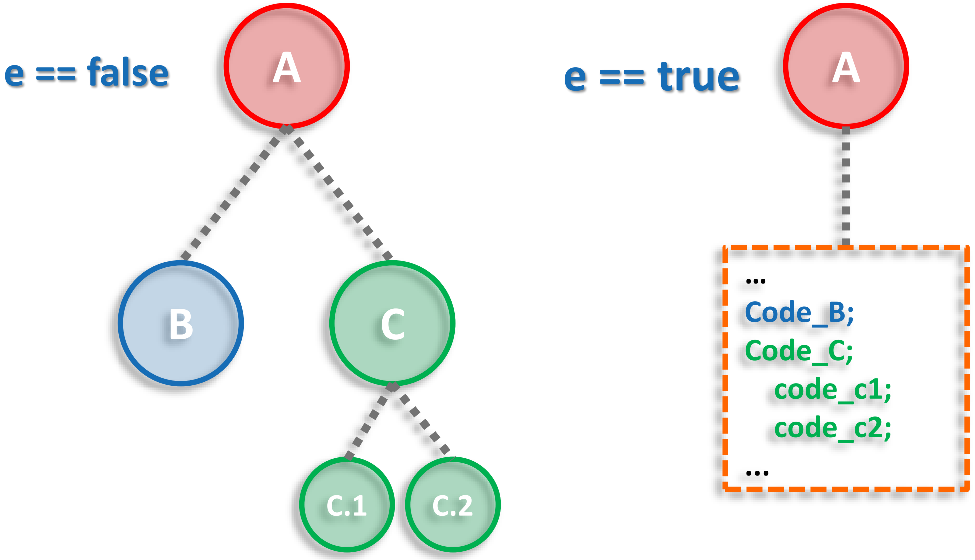
Really useful to debug tasking applications!

- Even if the `expression` is `false`, data-sharing clauses are honored

The `final` clause

- The `final(expression)` clause
  - → Nested tasks / recursive applications
  - → allows to avoid future task creation → reduces overhead but also reduces parallelism
- If the `expression` of the `final` clause evaluates to `true`
  - → The new task is created and executed normally but in its context all tasks will be executed immediately by the same thread (*included tasks*)

```
#pragma omp task final(e)
{
  #pragma omp task
  { … }
  #pragma omp task
  { … #C.1; #C.2 … }
  #pragma omp taskwait
}
```



- Data-sharing clauses are honored too!

The `mergeable` clause

- The `mergeable` clause
  - → Optimization: get rid of “data-sharing clauses are honored”
  - → This optimization can only be applied in *undeferred* or *included tasks*
- A Task that is annotated with the `mergeable` clause is called a *mergeable task*
  - → A task that may be a *merged task* if it is an *undeferred task* or an *included task*
- A *merged task* is:
  - → A task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region
- A good implementation could execute a merged task without adding any OpenMP-related overhead

Unfortunately, there are no OpenMP commercial implementations taking advantage of `final` neither `mergeable` =(

Programming OpenMP

NUMA

Christian Terboven

Michael Klemm

Improving Tasking Performance: Task Affinity

Motivation

- Techniques for process binding & thread pinning available
  - →OpenMP thread level: `OMP_PLACES & OMP_PROC_BIND`
  - →OS functionality: `taskset -c`

<u>OpenMP Tasking:</u>

- In general: Tasks may be executed by any thread in the team
  - →Missing task-to-data affinity may have detrimental effect on performance

<u>OpenMP 5.0:</u>

- `affinity` clause to express affinity to data

`affinity` clause

- New clause: `#pragma omp task affinity (list)`
  - →Hint to the runtime to execute task closely to physical data location
  - →Clear separation between dependencies and affinity
- Expectations:
  - →Improve data locality / reduce remote memory accesses
  - →Decrease runtime variability
- Still expect task stealing
  - →In particular, if a thread is under-utilized

Code Example

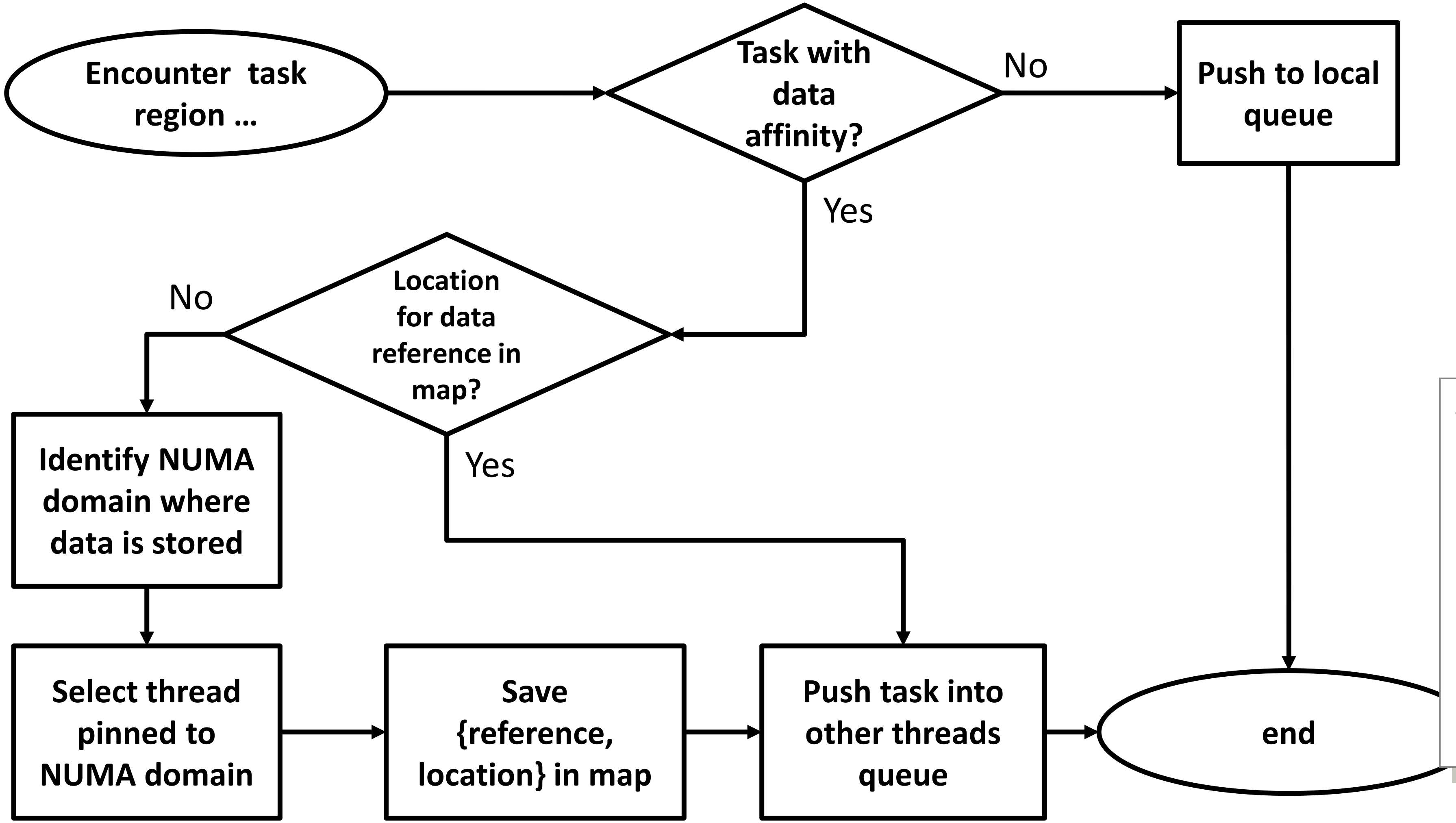
- Excerpt from task-parallel STREAM

```
#pragma omp task \
    shared(a, b, c, scalar) \
    firstprivate(tmp_idx_start, tmp_idx_end) \
    affinity( a[tmp_idx_start] )
{
   int i;
   for(i = tmp_idx_start; i <= tmp_idx_end; i++)
       a[i] = b[i] + scalar * c[i];
}
```

→Loops have been blocked manually (see `tmp_idx_start/end`)

→Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details

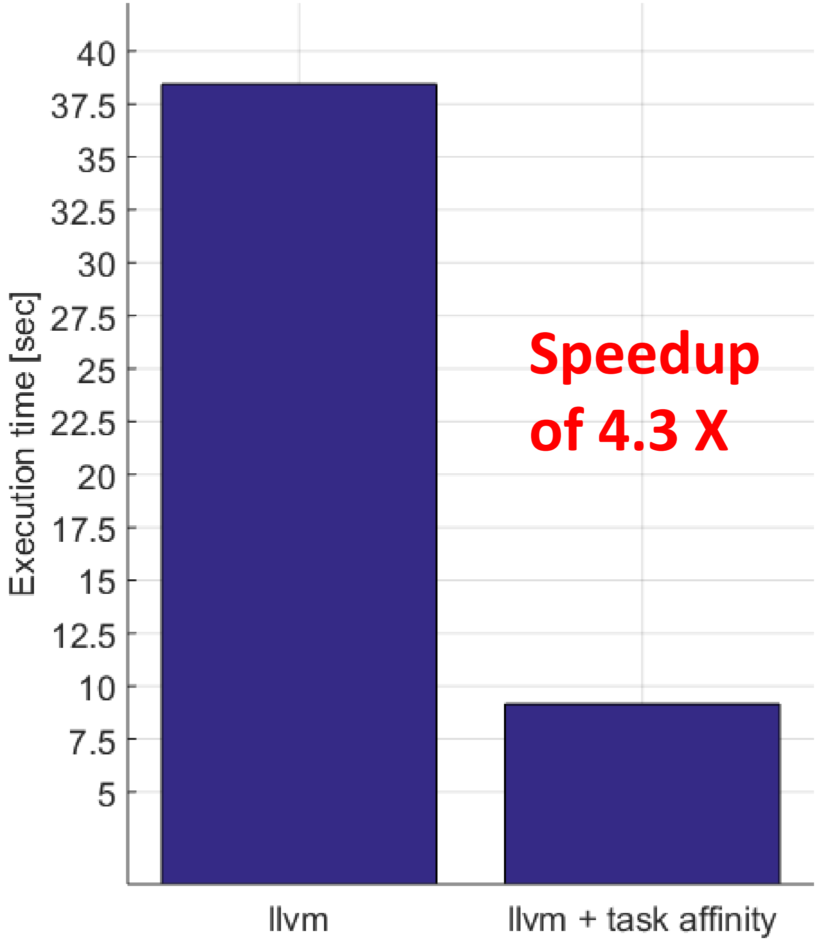


A map is introduced to store location information of data that was previously used

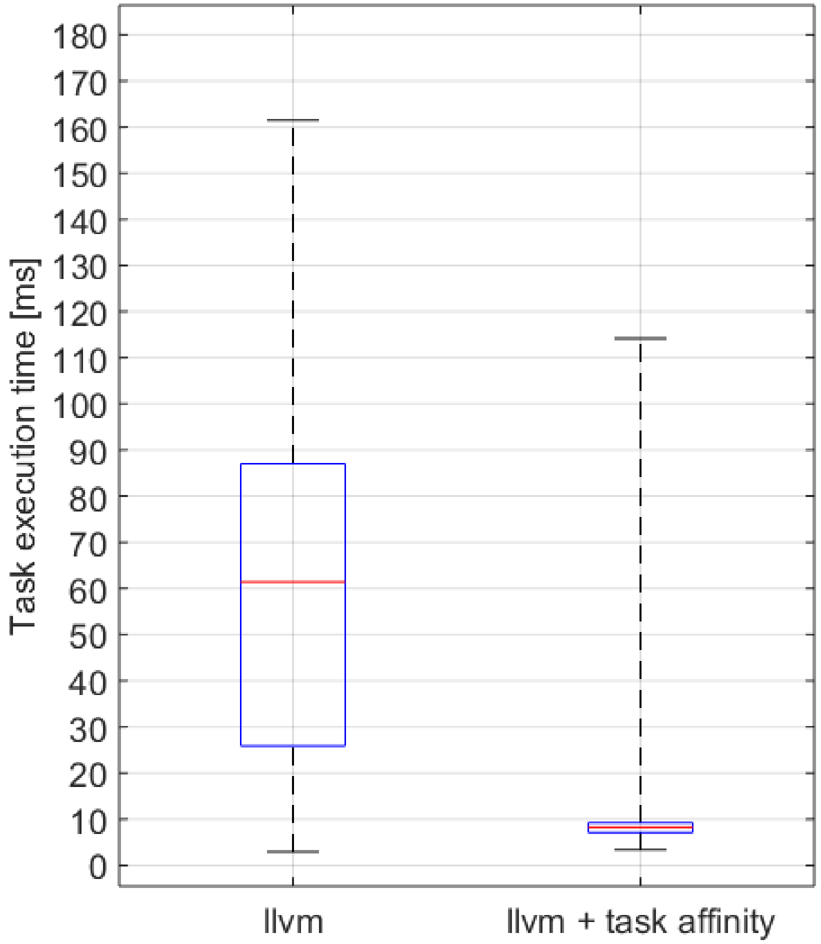
Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L. Olivier, and Matthias S. Müller. **Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime**. Proceedings of the 14th International Workshop on OpenMP, IWOMP 2018. September 26-28, 2018, Barcelona, Spain.

Evaluation

Program runtime
Median of 10 runs



Distribution of single task execution times



LIKWID: reduction of remote data volume from 69% to 13%

Summary

- Requirement for this feature: thread affinity enabled

- The `affinity` clause helps, if
  - →tasks access data heavily
  - →single task creator scenario, or task not created with data affinity
  - →high load imbalance among the tasks

- Different from thread binding: task stealing is absolutely allowed

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: HIP memory transfers

```
do_something();
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
do_something_else();
hipStreamSynchronize(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 hip_example() {
#pragma omp task      // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task // task B
    {
        do_something_else();
    }
    #pragma omp task // task C
    {
        hipStreamSynchronize(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 hip_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task                   // task B
    {
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        hipStreamSynchronize(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 callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ③ omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example() {
    omp_event_t *hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, hip_event, 0);
  ① }
#pragma omp task                     // task B
        do_something_else();

#pragma omp taskwait ② ④
#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 callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ②omp_fulfill_event((omp_event_t *) cb_data);
}
void hip_example() {
    omp_event_t *hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
  ①     hipStreamAddCallback(stream, callback, hip_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

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information

Programming OpenMP

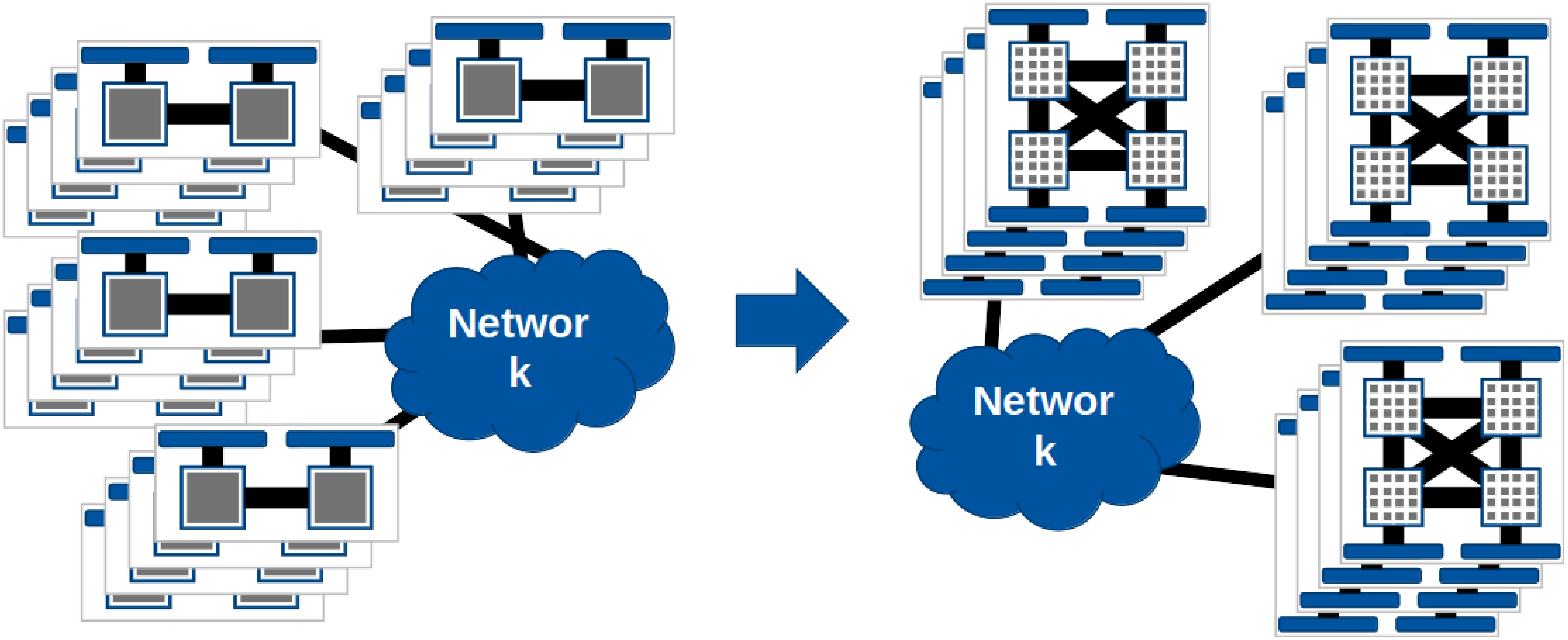
OpenMP and MPI

Christian Terboven
Michael Klemm

Motivation

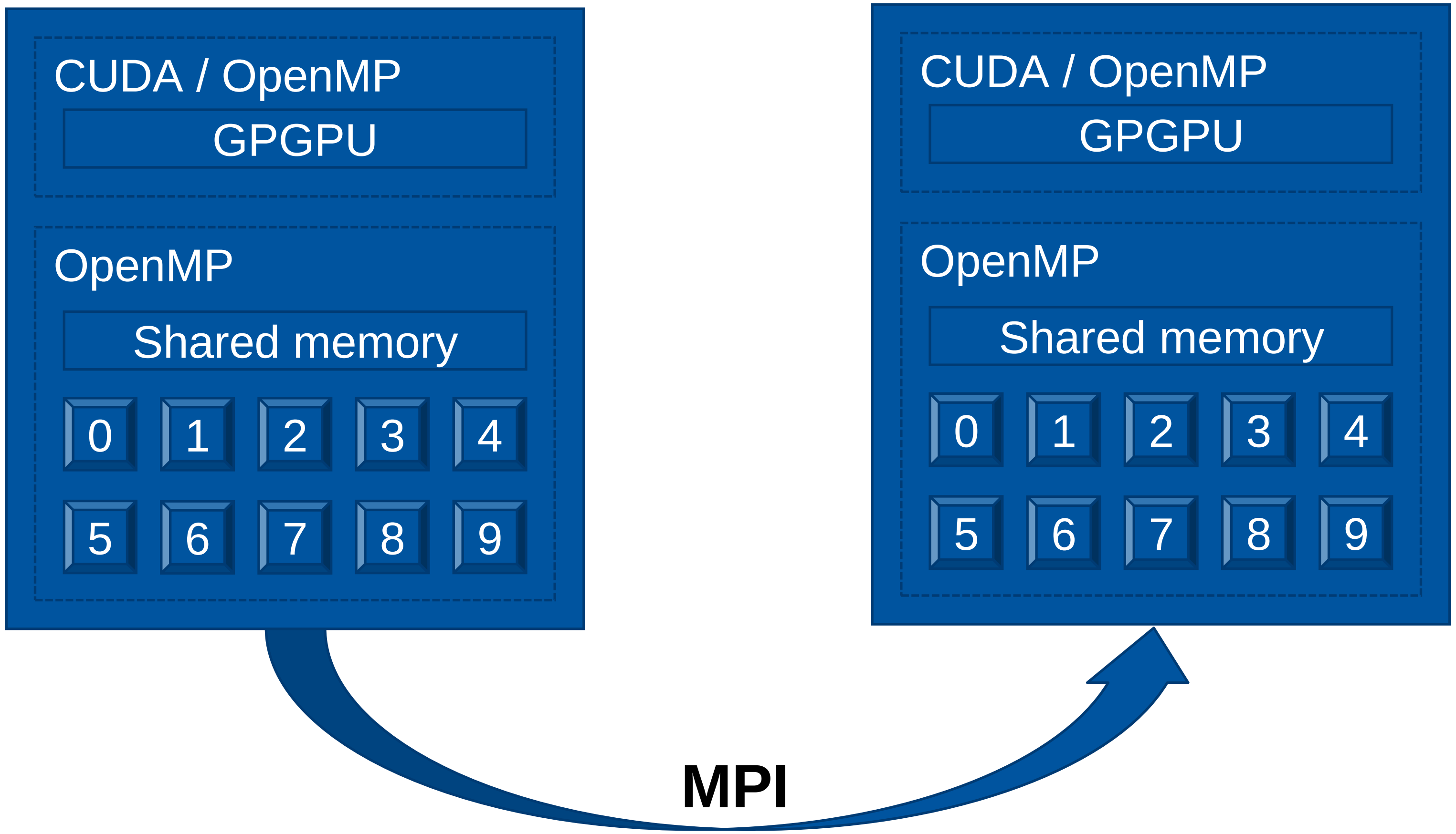
Motivation for hybrid programming

- 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

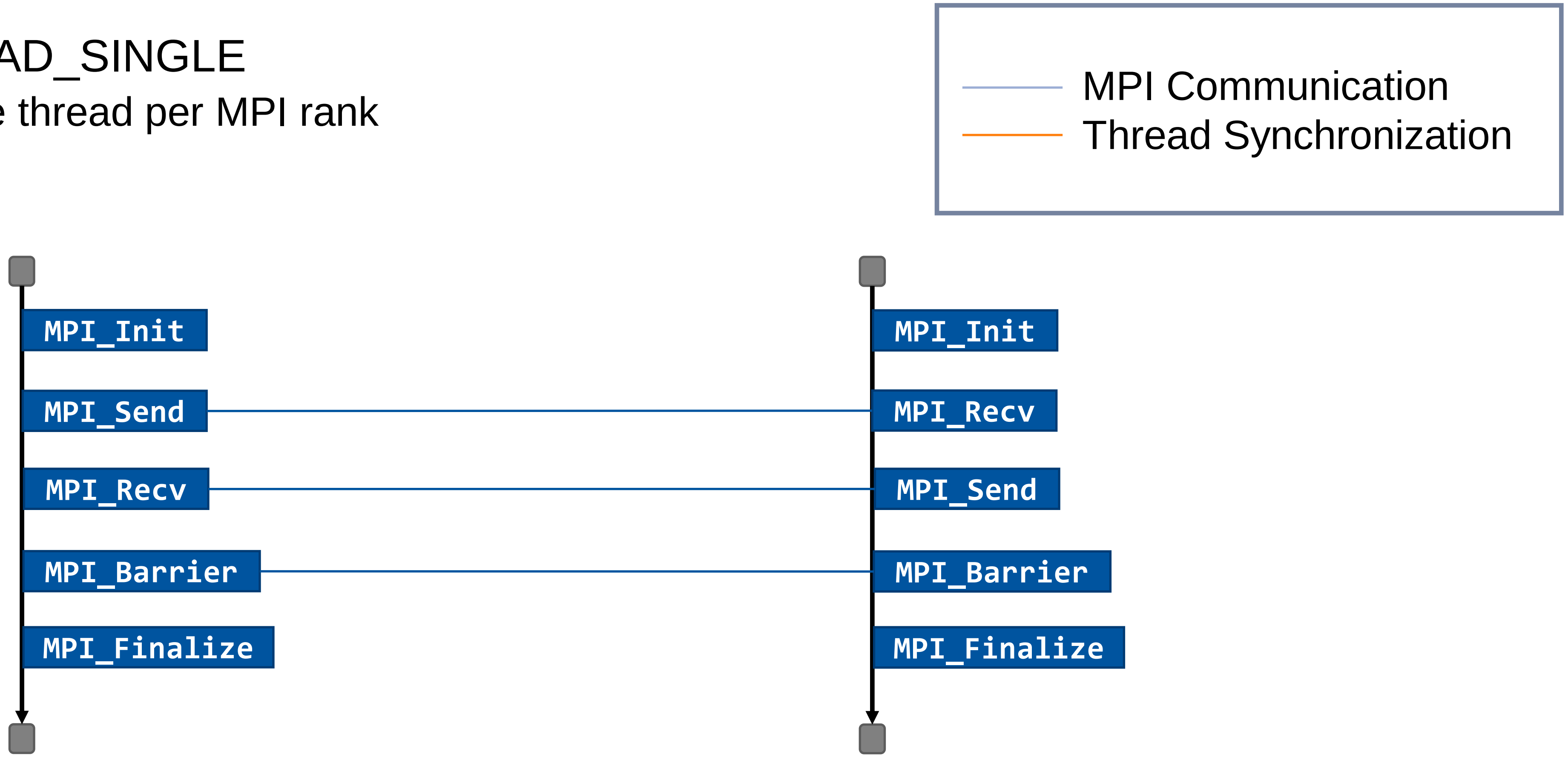
Higher levels

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

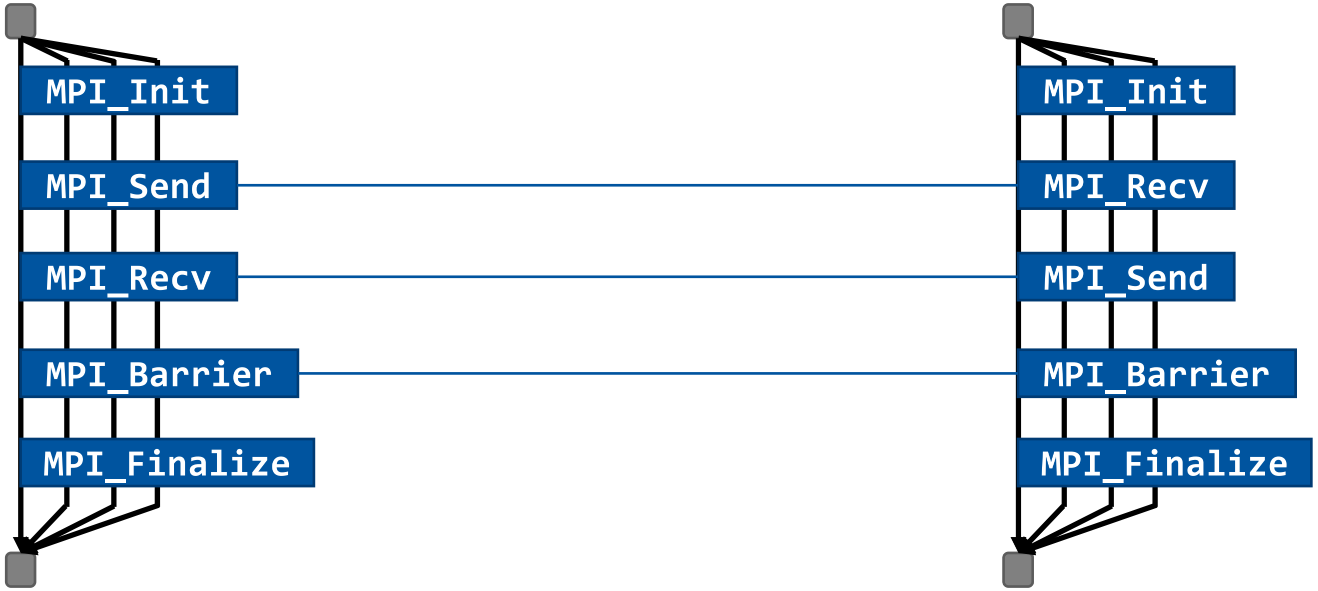
MPI – Threading support levels

- MPI_THREAD_SINGLE
  - Only one thread per MPI rank



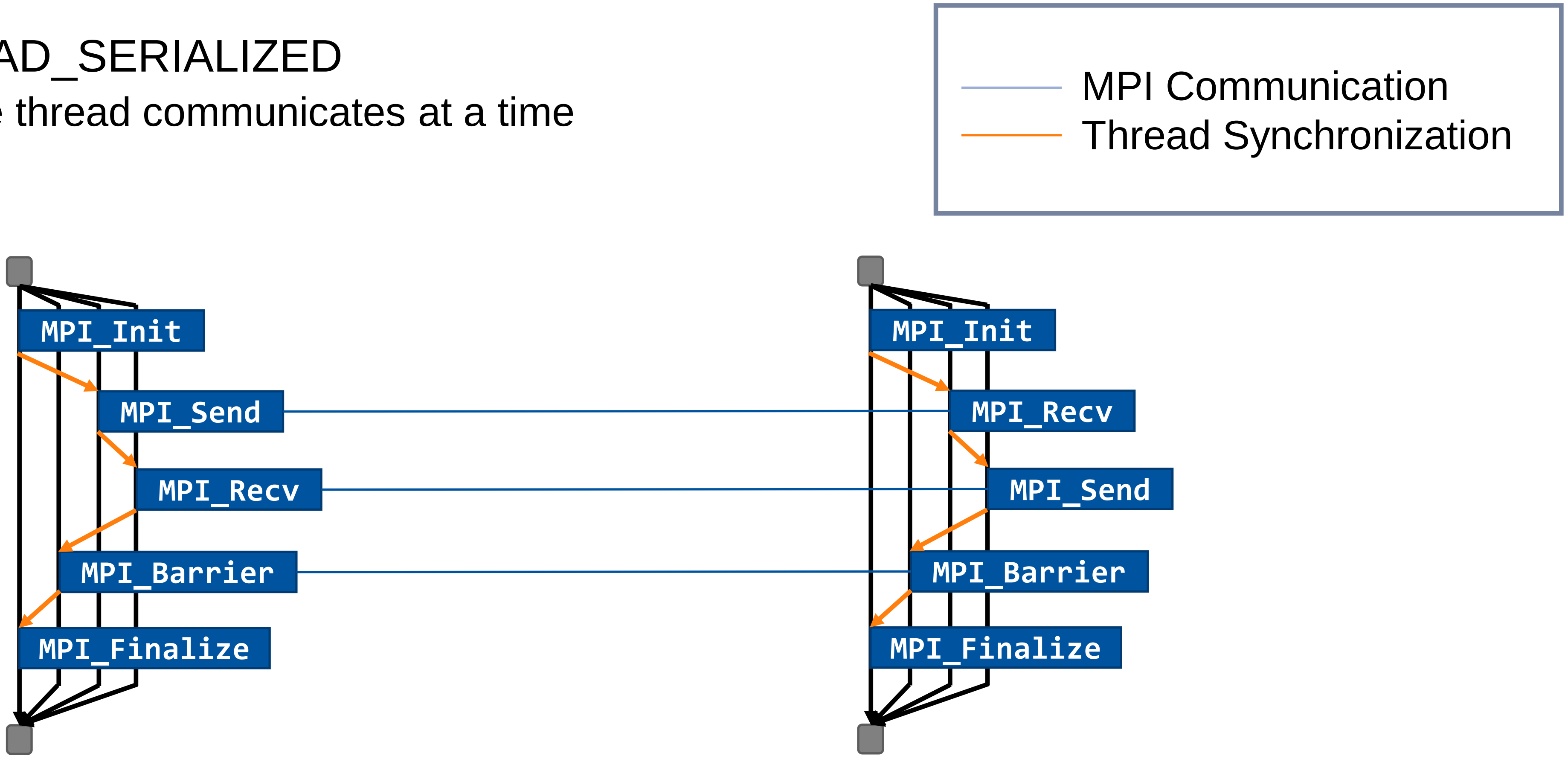
MPI – Threading support levels

- 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