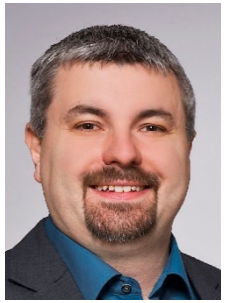
# Programming OpenMP

**Christian Terboven**

Michael Klemm



# Agenda (in total 7 Sessions)

- Session 1: OpenMP Introduction
- Session 2: Tasking
- **Session 3:** Optimization for **NUMA and SIMD**
  - →Review of Session 2 / homework assignments
  - →OpenMP and NUMA architectures
  - →Task Affinity
  - →SIMD
  - →Homework assignments ☺
- Session 4: What Could Possibly Go Wrong Using OpenMP
- Session 5: Introduction to Offloading with OpenMP
- Session 6: Advanced Offloading Topics
- Session 7: Selected / Remaining Topics

# Programming OpenMP

## *Review*

**Christian Terboven**
Michael Klemm

RWTH AACHEN UNIVERSITY
OpenMP

OpenMP

# *Questions?*

# *Fibonacci*

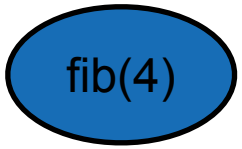
# Fibonacci illustrated

```
 1  int main(int argc,
 2           char* argv[])
 3  {
 4      [...]
 5      #pragma omp parallel
 6      {
 7          #pragma omp single
 8          {
 9              fib(input);
10          }
11      }
12      [...]
13.}
```

```
14  int fib(int n)   {
15      if (n < 2) return n;
16      int x, y;
17      #pragma omp task shared(x)
18      {
19          x = fib(n - 1);
20      }
21      #pragma omp task shared(y)
22      {
23          y = fib(n - 2);
24      }
25      #pragma omp taskwait
26          return x+y;
27.}
```

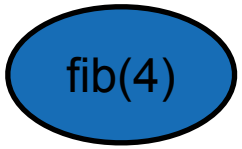
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

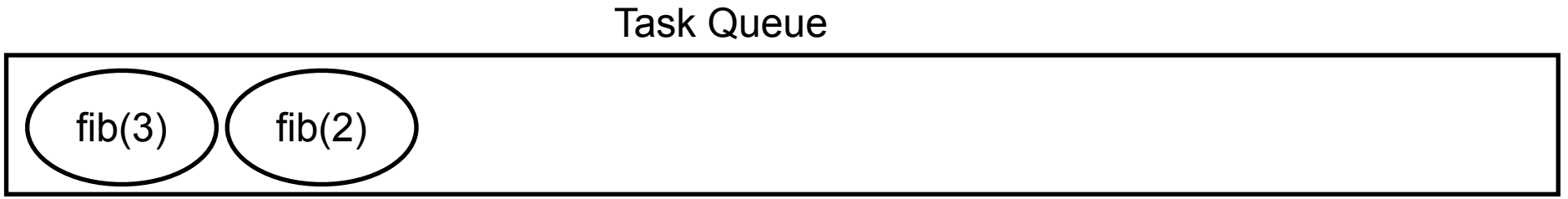
- **T1 enters fib(4)**




Task Queue

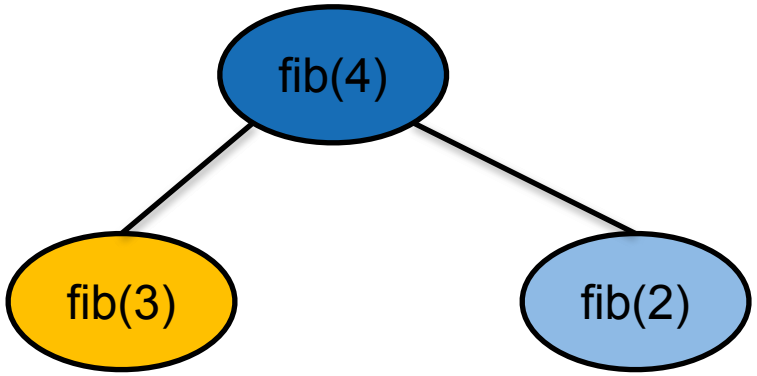
- T1 enters fib(4)
- **T1 creates tasks for fib(3) and fib(2)**



- T1 enters fib(4)

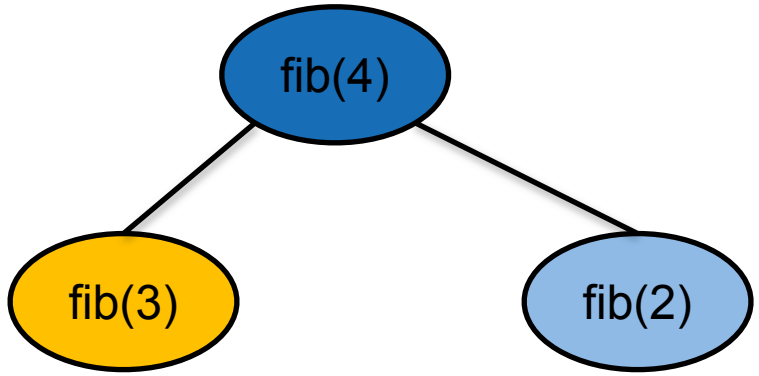
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**

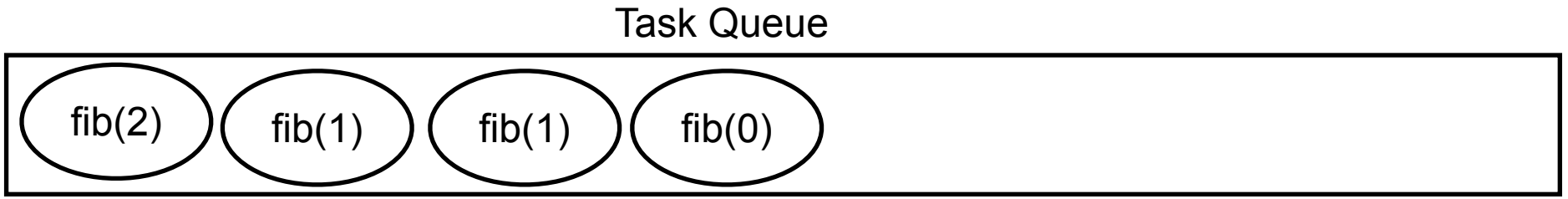



Task Queue

- T1 enters fib(4)

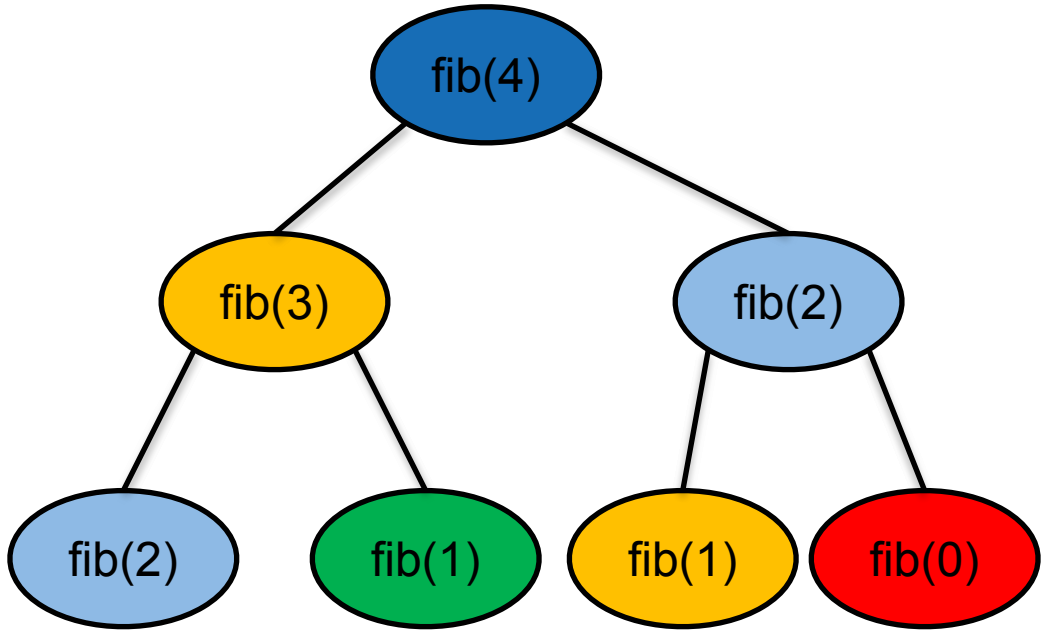
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks



- T1 enters fib(4)

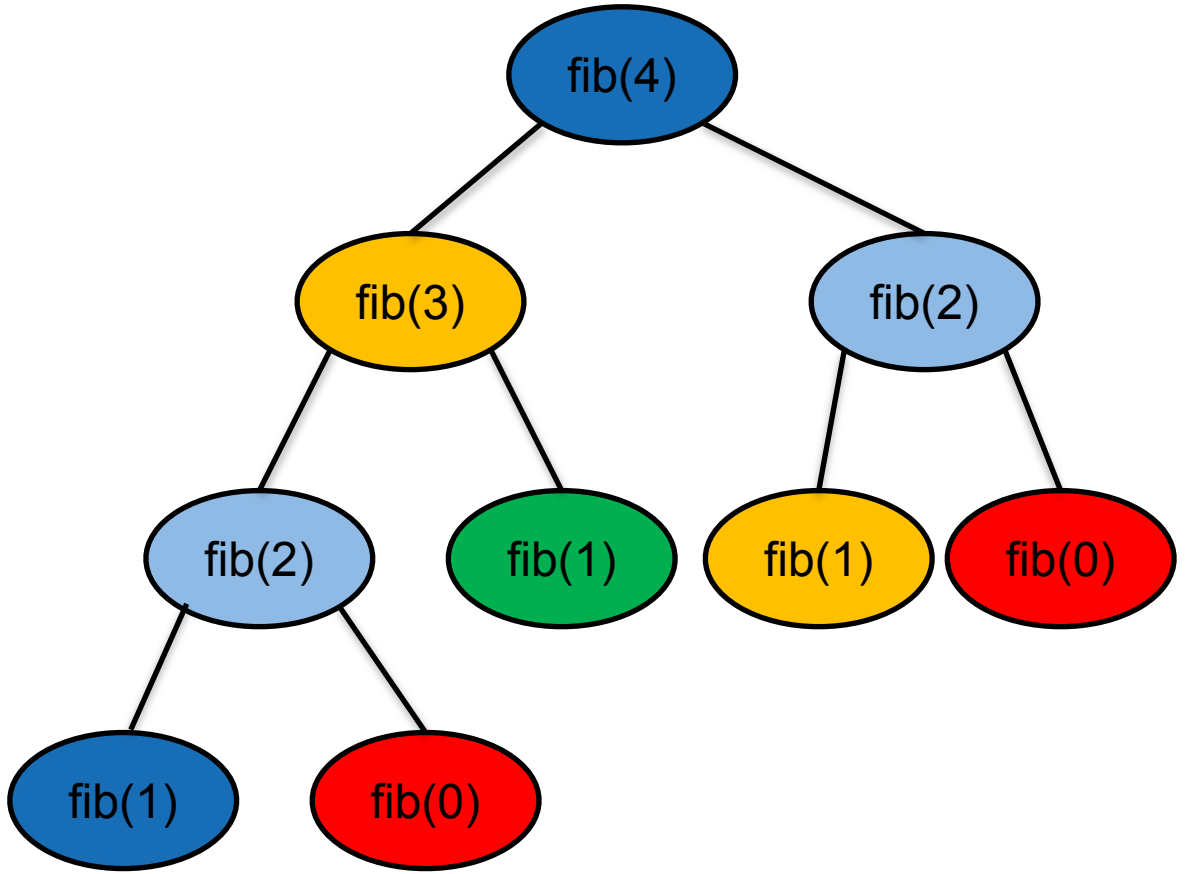
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks




Task Queue

- T1 enters fib(4)

- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



# For / Work-distribution

# Example solution: For w/ Tasking wo/ Red.

```
#pragma omp parallel firstprivate(presult)
{
#pragma omp single
{

        for (int i = 0; i < dimension; i++)
        {
#pragma omp task shared(presult)
{
                result += do_some_computation(i);
}
        }

} // end omp single

#pragma omp critical
{
        result += presult;
}

} // end omp parallel
```

# Example solution: For w/ Tasking

```
#pragma omp parallel reduction(task,+:result)
{
#pragma omp single
{

        for (int i = 0; i < dimension; i++)
        {
#pragma omp task in_reduction(+:result)
{
                result += do_some_computation(i);
}
        }

} // end omp single

} // end omp parallel
```

# Example solution: For w/ Taskloop

```
#pragma omp parallel reduction(task,+:result)
{
#pragma omp single
{

#pragma omp taskloop in_reduction(+:result)
        for (int i = 0; i < dimension; i++)
        {
                result += do_some_computation(i);
        }

} // end omp single

} // end omp parallel
```

# QuickSort

# Example solution: Quick Sort

# Example solution: Quick Sort

```
void quicksort(int * array, int first, int last){
        int pivotElement;
        if((last - first + 1) < 10000) {
                serial_quicksort(array, first, last);
        } else {
                pivotElement = pivot(array,first,last);
                #pragma omp task default(shared)
                {
                        quicksort(array,first,pivotElement-1);
                }
                #pragma omp task default(shared)
                {
                        quicksort(array,pivotElement+1,last);
                }
                #pragma omp taskwait
        }
}
```

# Programming OpenMP

## *Review*

**Christian Terboven**
Michael Klemm

# *Questions?*

# *Fibonacci*

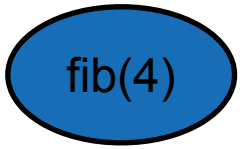
# Fibonacci illustrated

```
 1  int main(int argc,
 2           char* argv[])
 3  {
 4      [...]
 5      #pragma omp parallel
 6      {
 7          #pragma omp single
 8          {
 9              fib(input);
10          }
11      }
12      [...]
13.}
```

```
14  int fib(int n)   {
15      if (n < 2) return n;
16      int x, y;
17      #pragma omp task shared(x)
18      {
19          x = fib(n - 1);
20      }
21      #pragma omp task shared(y)
22      {
23          y = fib(n - 2);
24      }
25      #pragma omp taskwait
26          return x+y;
27.}
```

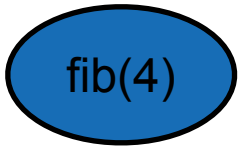
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

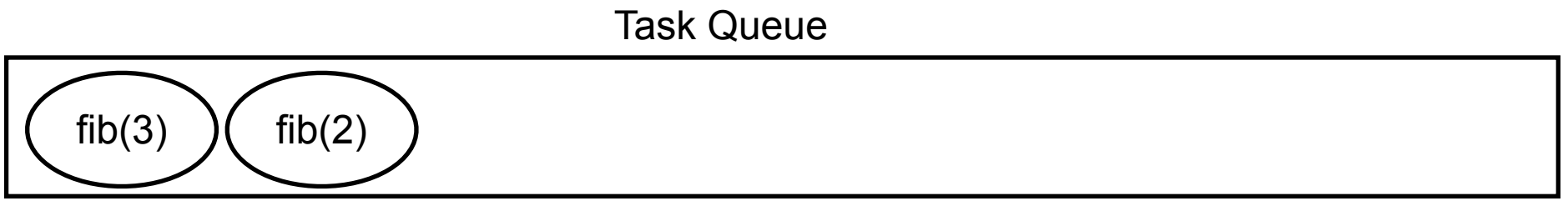
- **T1 enters fib(4)**




Task Queue

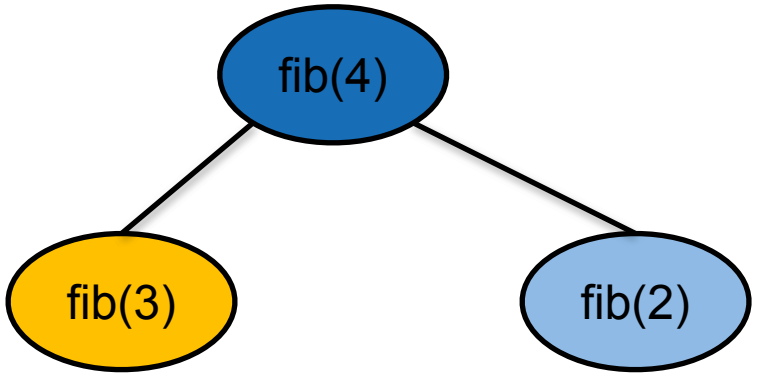
- T1 enters fib(4)
- **T1 creates tasks for fib(3) and fib(2)**



- T1 enters fib(4)

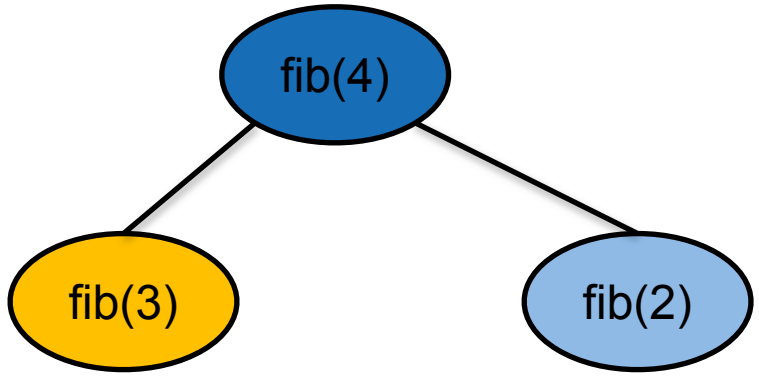
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**

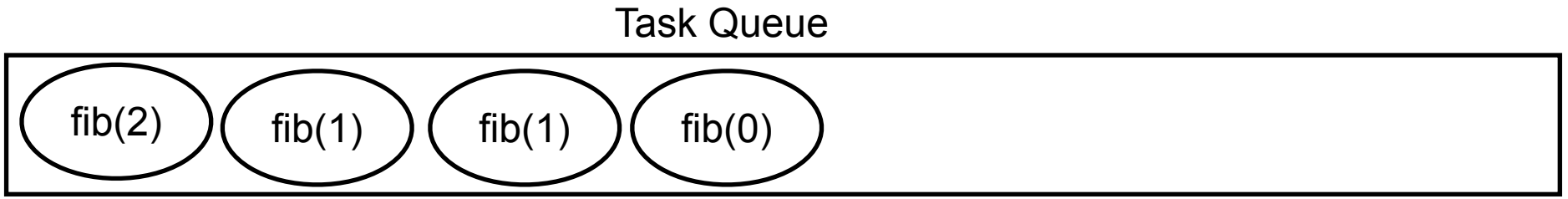



Task Queue

- T1 enters fib(4)

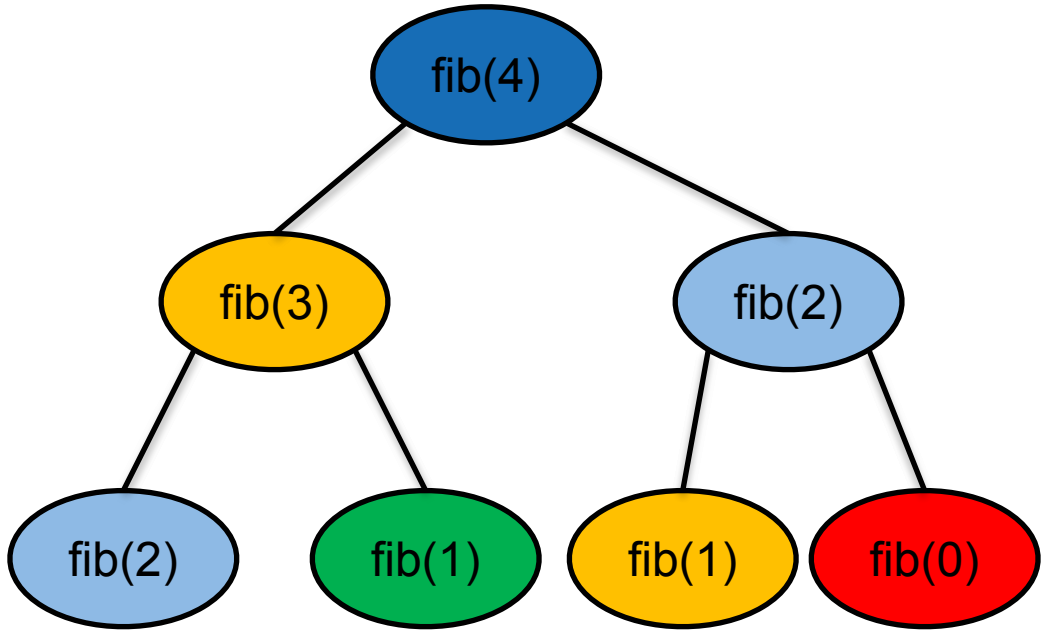
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks



- T1 enters fib(4)

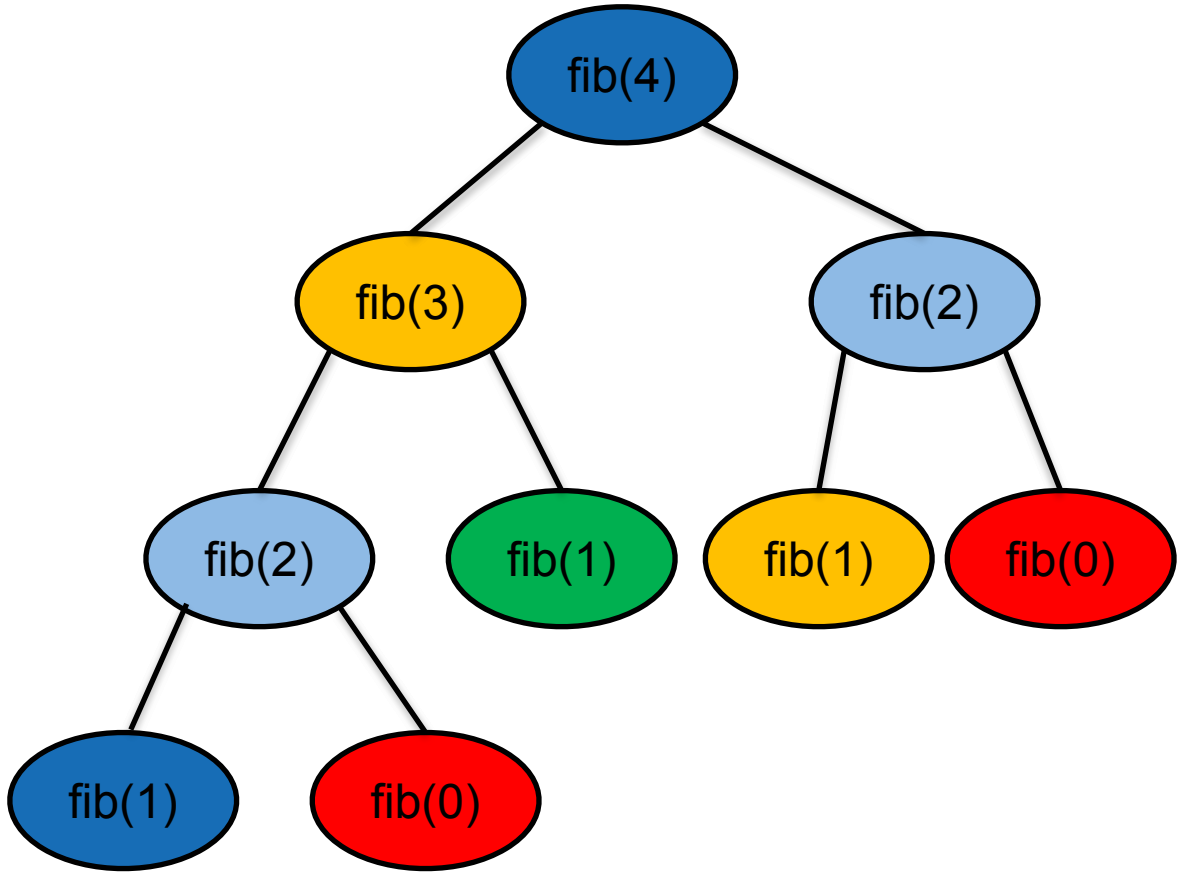
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks




Task Queue

- T1 enters fib(4)

- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



# For / Work-distribution

# Example solution: For w/ Tasking wo/ Red.

```
#pragma omp parallel firstprivate(presult)
{
#pragma omp single
{

        for (int i = 0; i < dimension; i++)
        {
#pragma omp task shared(presult)
{
                result += do_some_computation(i);
}
        }

} // end omp single

#pragma omp critical
{
        result += presult;
}

} // end omp parallel
```

# Example solution: For w/ Tasking

```
#pragma omp parallel reduction(task,+:result)
{
#pragma omp single
{

        for (int i = 0; i < dimension; i++)
        {
#pragma omp task in_reduction(+:result)
{
                result += do_some_computation(i);
}
        }

} // end omp single

} // end omp parallel
```

# Example solution: For w/ Taskloop

```
#pragma omp parallel reduction(task,+:result)
{
#pragma omp single
{

#pragma omp taskloop in_reduction(+:result)
        for (int i = 0; i < dimension; i++)
        {
                result += do_some_computation(i);
        }

} // end omp single

} // end omp parallel
```

# QuickSort

# Programming OpenMP

***NUMA***

**Christian Terboven**

Michael Klemm

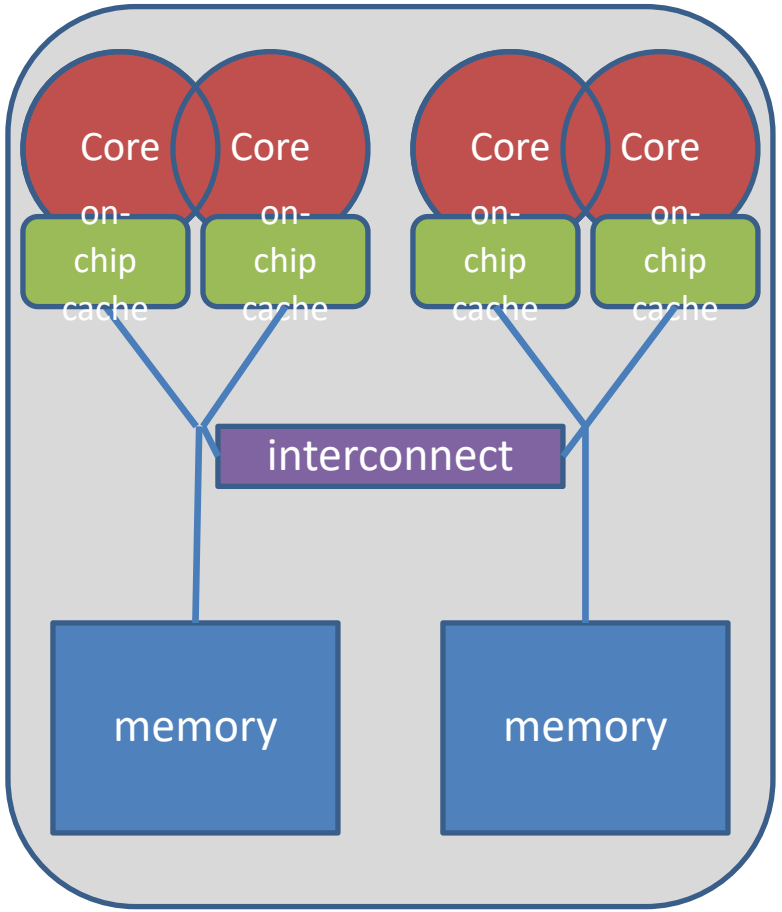
# Review: NUMA concept

# Non-uniform Memory

## *How To Distribute The Data ?*

```
double* A;

A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {

   A[i] = 0.0;

}
```

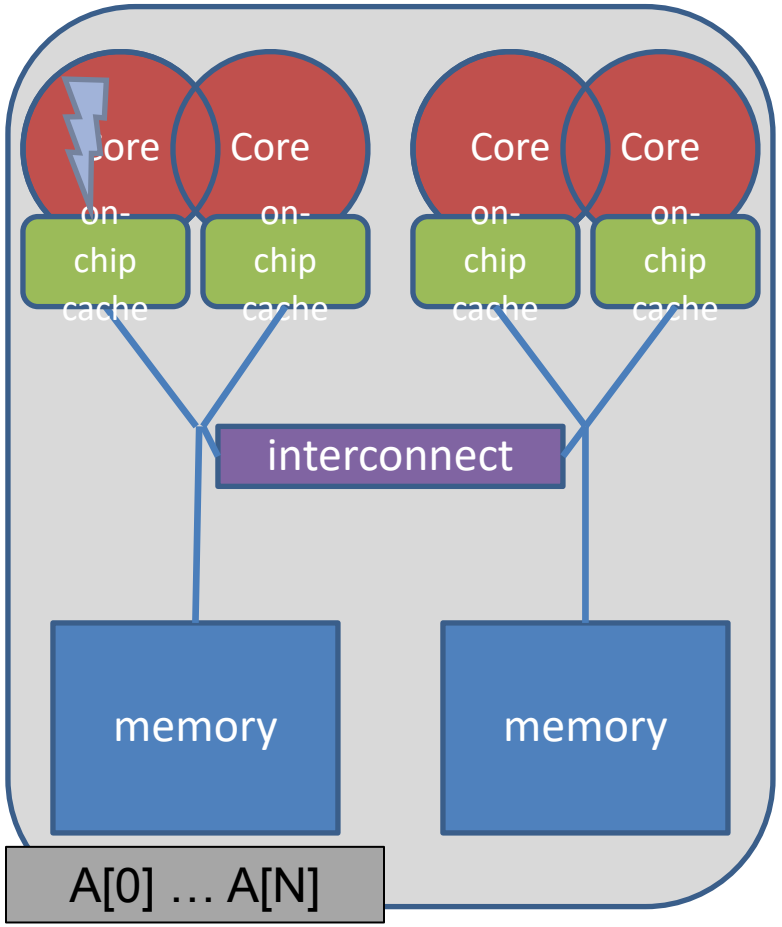


# Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

```
double* A;

A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {

   A[i] = 0.0;

}
```



# About Data Distribution

- Important aspect on cc-NUMA systems
  - →If not optimal, longer memory access times and hotspots

- Placement comes from the Operating System
  - →This is therefore Operating System dependent

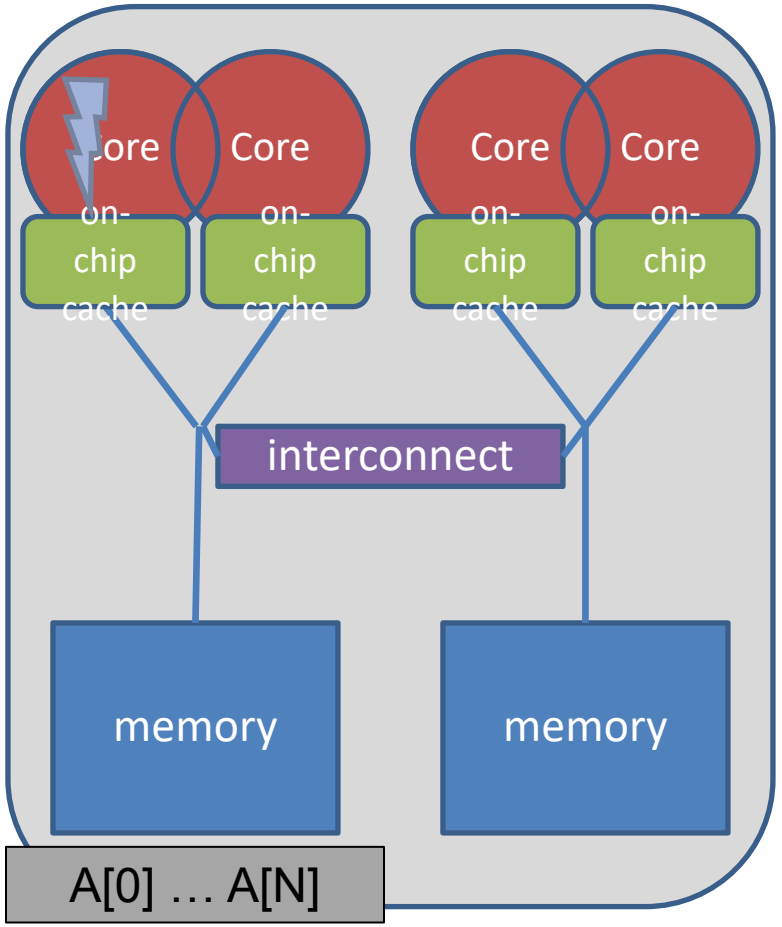
- Windows, Linux and Solaris all use the “First Touch” placement policy by default
  - →May be possible to override default (check the docs)

# Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

```
double* A;

A = (double*)
    malloc(N * sizeof(double));
```

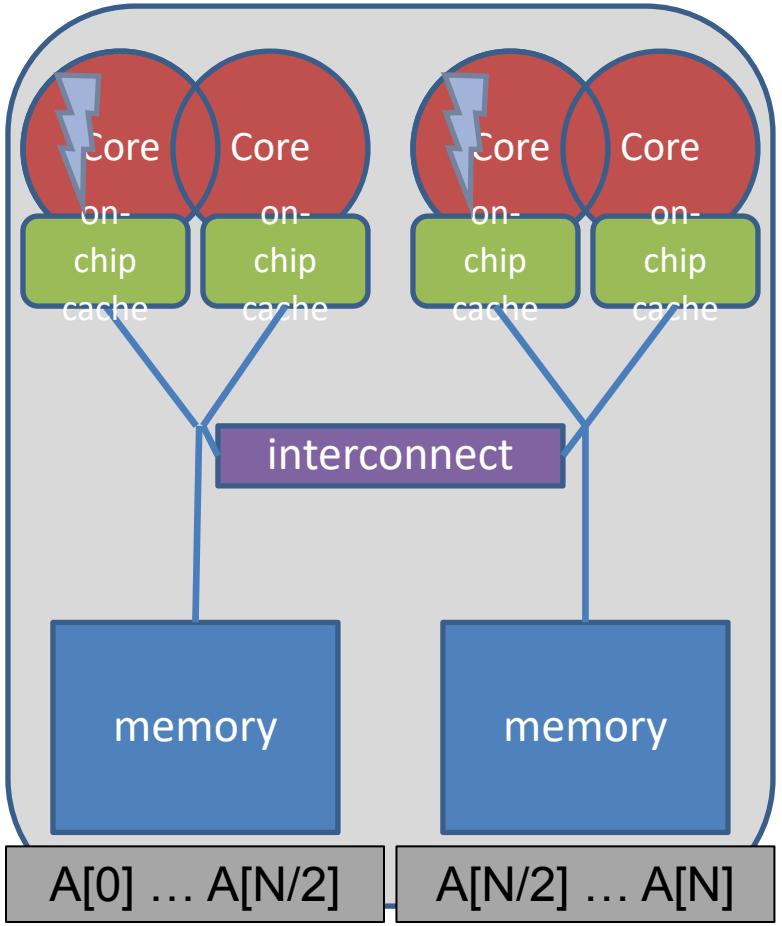
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



# First Touch Memory Placement

- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition**

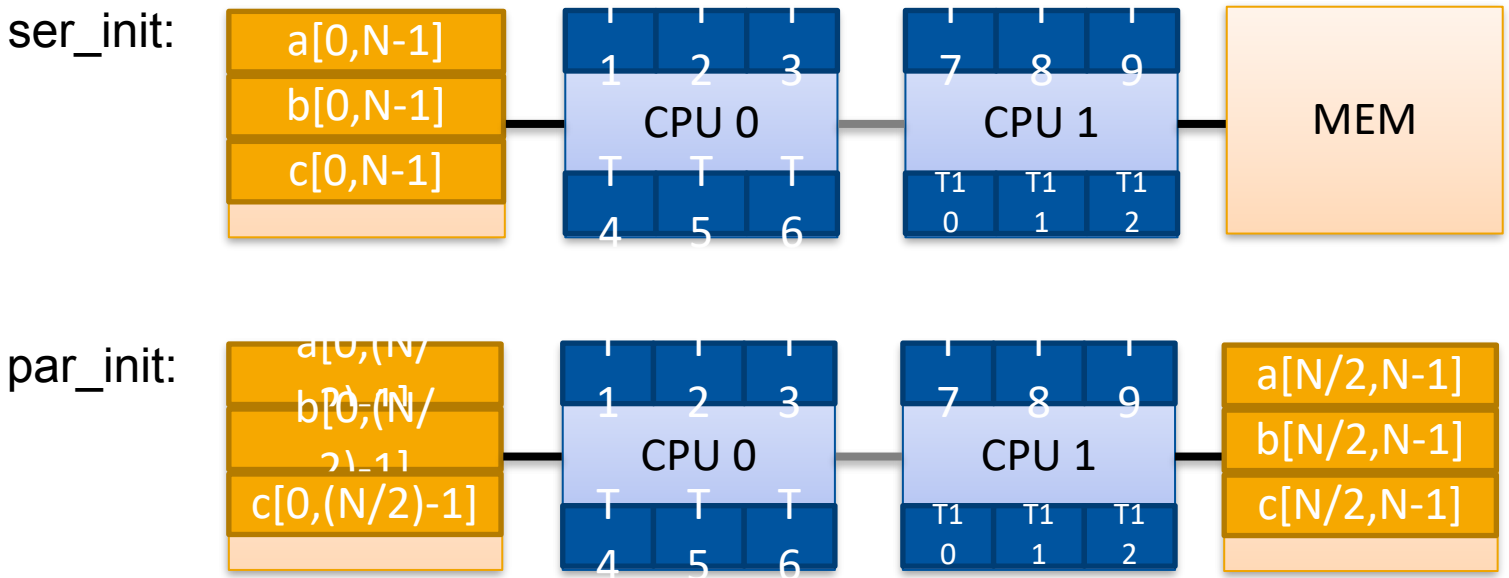
```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



# Serial vs. Parallel Initialization

- Stream example on 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads:

| | copy | scale | add | triad |
|---|---|---|---|---|
| ser_init | 18.8 GB/s | 18.5 GB/s | 18.1 GB/s | 18.2 GB/s |
| par_init | 41.3 GB/s | 39.3 GB/s | 40.3 GB/s | 40.4 GB/s |



# *Thread Binding and Memory Placement*

# Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:
  - → Intel MPI's `cpuinfo` tool
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor ids to cpu cores that the OS uses.
  - → hwlocs' `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a (graphical) representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids to cpu cores that the OS uses and additional info on caches.

# Decide for Binding Strategy

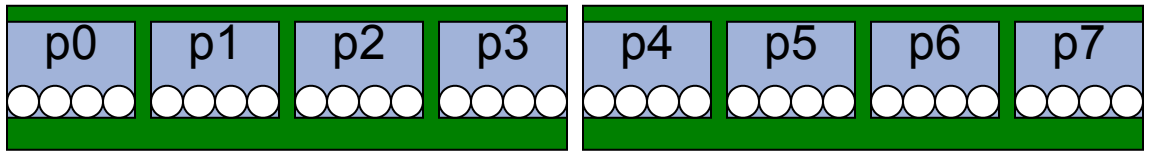
- Selecting the „right“ binding strategy depends not only on the topology, but also on application characteristics.
  - → Putting threads far apart, i.e., on different sockets
    - → May improve aggregated memory bandwidth available to application
    - → May improve the combined cache size available to your application
    - → May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - → May improve performance of synchronization constructs
    - → May decrease the available memory bandwidth and cache size

# Places + Binding Policies (1/2)

- Define OpenMP Places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e. `OMP_PLACES=cores`
- Define a set of OpenMP Thread Affinity Policies
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near primary thread
  - → PRIMARY: collocate OpenMP thread with primary thread
- Goals
  - → user has a way to specify where to execute OpenMP threads
  - → locality between OpenMP threads / less false sharing / memory bandwidth

# OMP_PLACES env. variable

- Assume the following machine:




→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

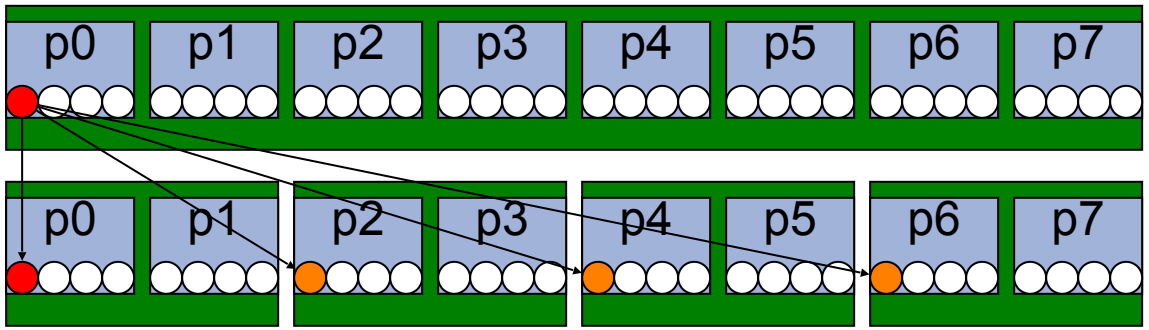
- Abstract names for OMP_PLACES:
  - → threads: Each place corresponds to a single hardware thread.
  - → cores: Each place corresponds to a single core (having one or more hardware threads).
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores).
  - → ll_caches (5.1): Each place corresponds to a set of cores that share the last level cache.
  - → numa_domains (5.1): Each places corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

# OpenMP 4.0: Places + Policies

- Example‘s Objective:
  - → separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)
  - → spread creates partition, compact binds threads within respective partition

```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8    = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4

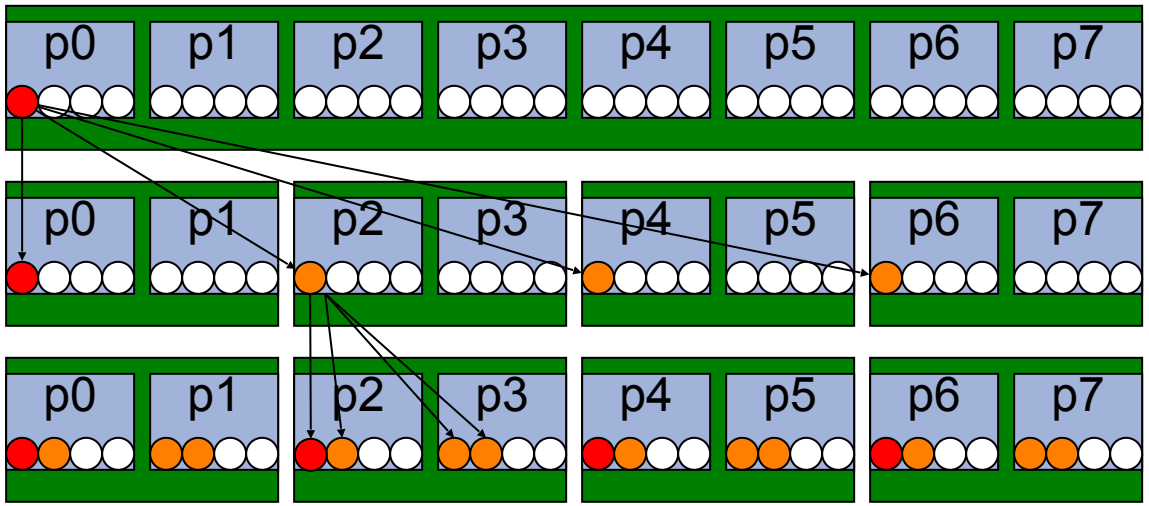


# OpenMP 4.0: Places + Policies

- Example‘s Objective:
  - → separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)
  - → spread creates partition, compact binds threads within respective partition

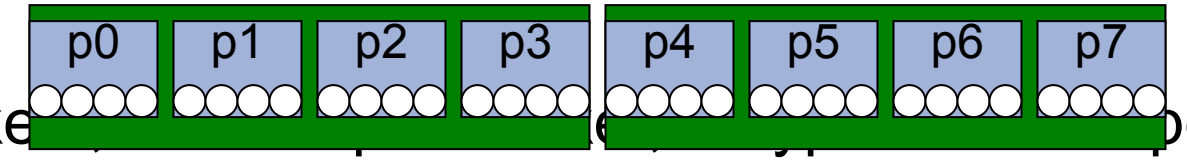
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8    = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4



# More Examples (1/3)

- Assume the following machine:




→2 socke per core

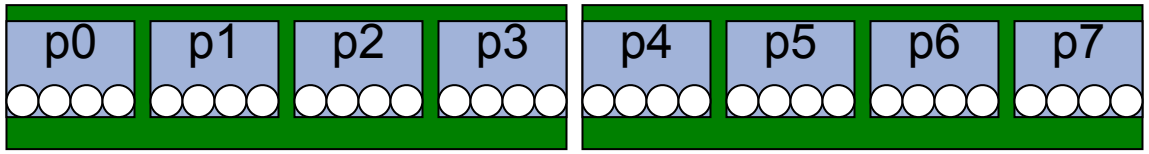
- Parallel Region with two threads, one per socket

→`OMP_PLACES=sockets`

→`#pragma omp parallel num_threads(2) proc_bind(spread)`

# More Examples (2/3)

- Assume the following machine:




- Parallel Region with four threads, one per core, but only on the first socket
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(4) proc_bind(close)`

# More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(2) proc_bind(spread)`
  - → `#pragma omp parallel num_threads(4) proc_bind(close)`

- Places API routines allow to
  - → query information about binding…
  - → query information about the place partition…

# Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

# Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

# OpenMP 5.0 way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - → Instructs the runtime to display formatted affinity information
  - → Example output for two threads on two physical cores:
  - → Output can be formatted with `OMP_AFFINITY_FORMAT` env var or corresponding routine
  - → Formatted affinity information can be printed with `omp_display_affinity(const char* format)`

# OpenMP 5.0 way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - → Instructs the runtime to display formatted affinity information
  - → Example output for two threads on two physical cores:

```
nesting_level=   1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=   2,3
```

  - → Output can be formatted with `OMP_AFFINITY_FORMAT` or the corresponding routine
  - → Formatted affinity information can be printed with

```
omp_display_affinity(const char* format)
```

# Affinity format specification

| | | | |
|---|---|---|---|
| t | omp_get_team_num() | a | omp_get_ancestor_thread_num() at level-1 |
| T | omp_get_num_teams() | H | hostname |
| L | omp_get_level() | P | process identifier |
| n | omp_get_thread_num() | i | native thread identifier |
| N | omp_get_num_threads() | A | thread affinity: list of processors (cores) |

- Example:

→Possible output:

# Affinity format specification

| | | | |
|---|---|---|---|
| t | omp_get_team_num() | a | omp_get_ancestor_thread_num() at level-1 |
| T | omp_get_num_teams() | H | hostname |
| L | omp_get_level() | P | process identifier |
| n | omp_get_thread_num() | i | native thread identifier |
| N | omp_get_num_threads() | A | thread affinity: list of processors (cores) |

- Example:

```
OMP_AFFINITY_FORMAT="Affinity: %0.3L %.8n %.15{A} %.12H"
```

→Possible output:

```
Affinity: 001          0       0-1,16-17       host003
Affinity: 001          1       2-3,18-19       host003
```

# Fine-grained control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - → By carefully touching data by the thread which later uses it
  - → By changing the default memory allocation strategy
    - → Linux: `numactl` command
  - → By explicit migration of memory pages
    - → Linux: `move_pages()`

- Example: using numactl to distribute pages round-robin:
  - → `numactl -interleave=all ./a.out`

# Managing Memory Spaces

# Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- …

**Cascade Lake (Leonide at INRIA)**

```
CPU: Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Freq Govenor: performance
---------------------
available: 4 nodes (0-3)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18
             20 22 24 26 28 30 32 34 36 38
node 0 size: 191936 MB
node 0 free: 178709 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23
             25 27 29 31 33 35 37 39
node 1 size: 192016 MB
node 1 free: 179268 MB
node 2 cpus:
node 2 size: 759808 MB
node 2 free: 759794 MB
node 3 cpus:
node 3 size: 761856 MB
node 3 free: 761851 MB
node distances:
node   0   1   2   3
  0:  10  21  17  28
  1:  21  10  28  17
  2:  17  28  10  28
  3:  28  17  28  10
```

**DRAM + Optane**

# Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
- OpenMP allocators are of type `omp_allocator_handle_t`
- Default allocator for Host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API
- OpenMP 5.0 supports a set of memory allocators

# OpenMP allocators

- Selection of a certain kind of memory

| Allocator name | Storage selection intent |
|---|---|
| omp_default_mem_alloc | use default storage |
| omp_large_cap_mem_alloc | use storage with large capacity |
| omp_const_mem_alloc | use storage optimized for read-only variables |
| omp_high_bw_mem_alloc | use storage with high bandwidth |
| omp_low_lat_mem_alloc | use storage with low latency |
| omp_cgroup_mem_alloc | use storage close to all threads in the contention group of the thread requesting the allocation |
| omp_pteam_mem_alloc | use storage that is close to all threads in the same parallel region of the thread requesting the allocation |
| omp_thread_local_mem_alloc | use storage that is close to the thread requesting the allocation |

# Using OpenMP Allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`
  - → `allocator` argument is optional
- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

# OpenMP allocator traits / 1

- Allocator traits control the behavior of the allocator

| | |
|---|---|
| sync_hint | contended, uncontended, serialized, private<br>default: contended |
| alignment | positive integer value that is a power of two<br>default: 1 byte |
| access | all, cgroup, pteam, thread<br>default: all |
| pool_size | positive integer value |
| fallback | default_mem_fb, null_fb, abort_fb, allocator_fb<br>default: default_mem_fb |
| fb_data | an allocator handle |
| pinned | true, false<br>default: false |
| partition | environment, nearest, blocked, interleaved<br>default: environment |

# OpenMP Allocator Traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - → `default_mem_fb`: return system's default memory
  - → Other options: null, abort, or use different allocator
- `pinned`: request pinned memory, i.e. for GPUs

# OpenMP Allocator Traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - → `environment`: use system's default behavior
  - → `nearest`: most closest memory
  - → `blocked`: partitioning into approx. same size with at most one block per storage resource
  - → `interleaved`: partitioning in a round-robin fashion across the storage resources

# OpenMP Allocator Traits / 4

- Construction of allocators with traits via
  - →
```
omp_allocator_handle_t   omp_init_allocator(
  omp_memspace_handle_t memspace,
  int ntraits, const omp_alloctrait_t traits[]);
```
  - →Selection of memory space mandatory
  - →Empty traits set: use defaults
- Allocators have to be destroyed with `*_destroy_*`
- Custom allocator can be made default with `omp_set_default_allocator(omp_allocator_handle_t allocator)`

# OpenMP Memory Spaces

- Storage resources with explicit support in OpenMP:

| | |
|---|---|
| omp_default_mem_space | System's default memory resource |
| omp_large_cap_mem_space | Storage with larg(er) capacity |
| omp_const_mem_space | Storage optimized for variables with constant value |
| omp_high_bw_mem_space | Storage with high bandwidth |
| omp_low_lat_mem_space | Storage with low latency |

→Exac

→Pre-d

# Memory Management Status

- **LLVM OpenMP runtime internally already uses libmemkind (libnuma, numactl)**
  - → Support for various kinds of memory: DDR, HBW and Persistent Memory (Optane)
  - → Library loaded at initialization (checks for availability)
  - → If requested memory space for allocator is not available → fallback to DDR

- **Memory Management implementation in LLVM still not complete**
  - → Some allocator traits not implemented yet
  - → Some `partition` values not implemented yet (environment, interleaved, nearest, blocked)
  - → Semantics of `omp_high_bw_mem_space and omp_large_cap_mem_space` unclear. Which memory should be used?
    - → Explicitly target HBM → currently implemented in LLVM

- **LLVM has custom implementation of aligned memory allocation**
  - → Allocation covers → {Allocator Information + Requested Size + Buffer based on alignment}

# Programming OpenMP

***NUMA***

**Christian Terboven**
Michael Klemm

RWTH AACHEN UNIVERSITY
OpenMP

# 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`

OpenMP Tasking:

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

OpenMP 5.0:

- `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

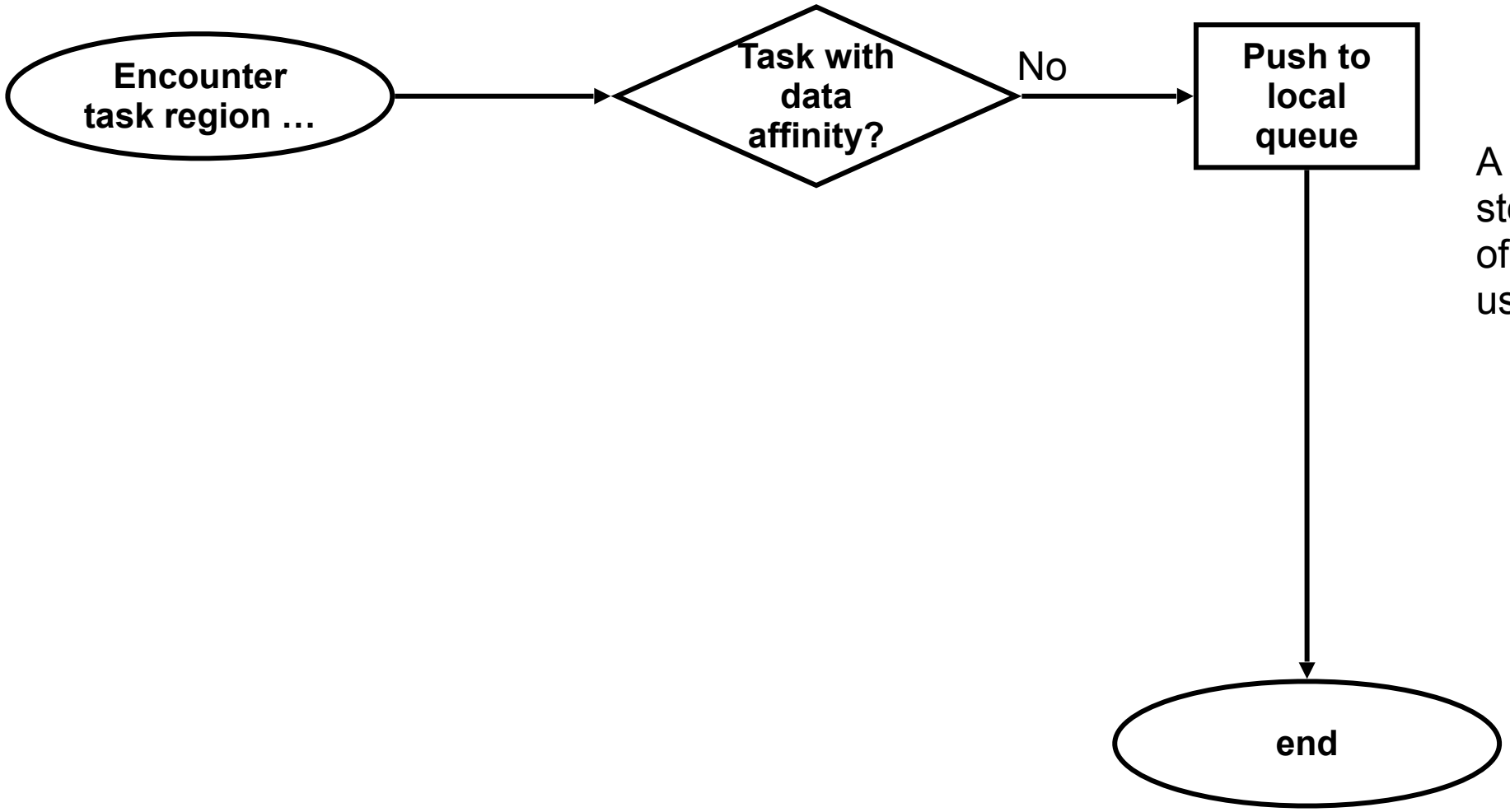
- Excerpt from task-parallel STREAM

```
1   #pragma omp task \
2       shared(a, b, c, scalar) \
3       firstprivate(tmp_idx_start, tmp_idx_end) \
4       affinity( a[tmp_idx_start] )
5   {
6      int i;
7      for(i = tmp_idx_start; i <= tmp_idx_end; i++)
8           a[i] = b[i] + scalar * c[i];
9   }
```

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

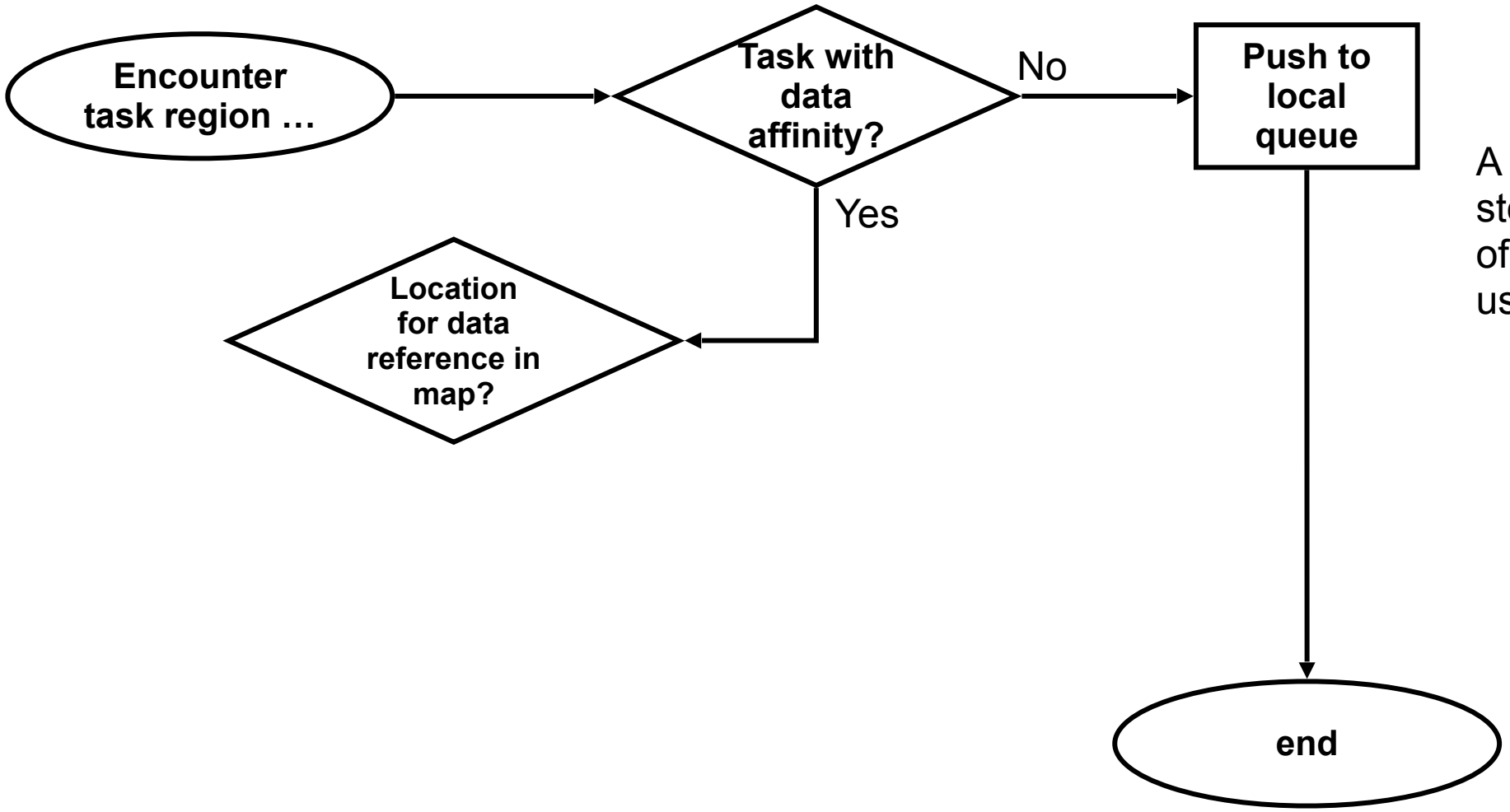
→Assumption: initialization and computation have same blocking and same affinity

# Selected LLVM implementation details



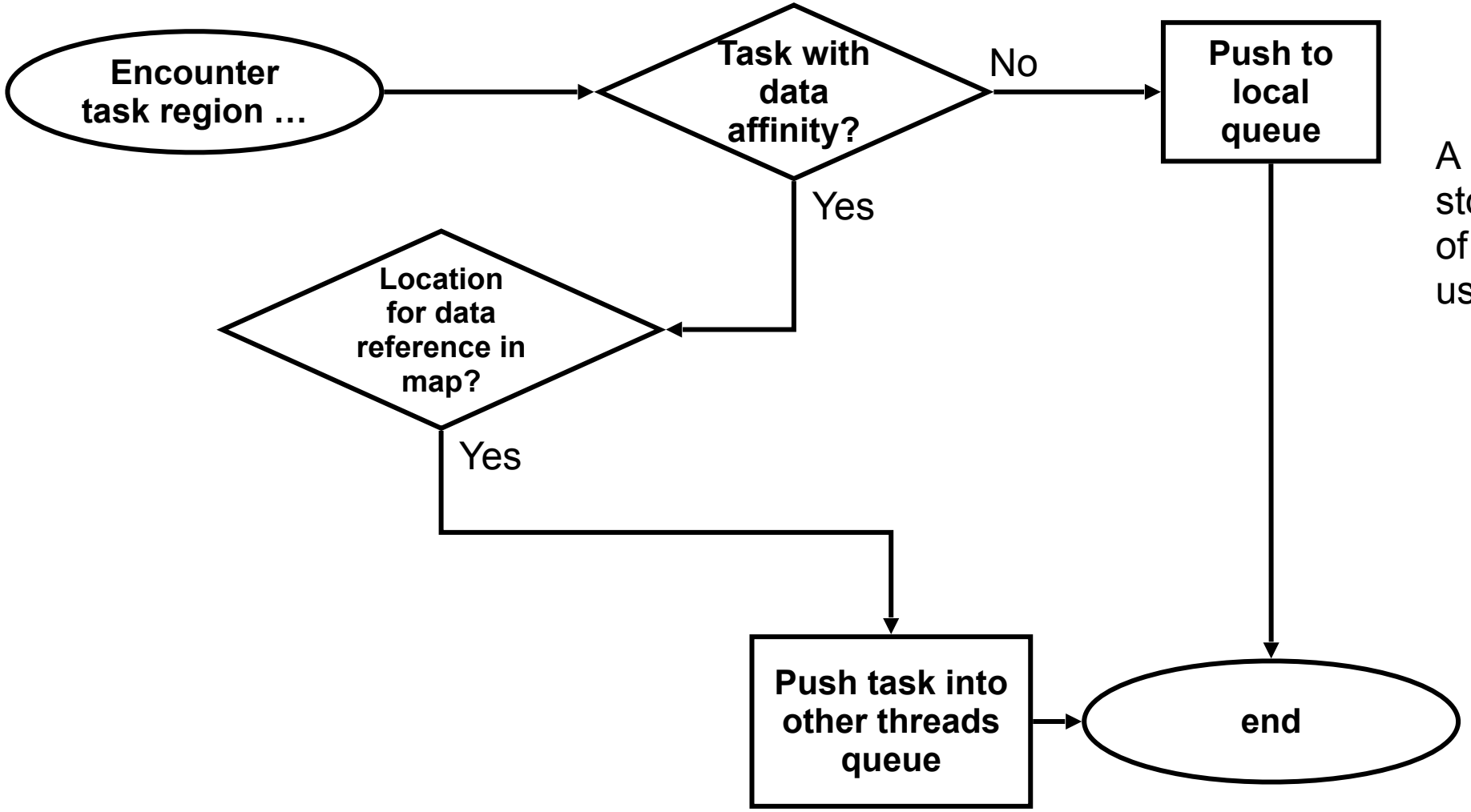

A map is introduced to store location information of data that was previously used

# Selected LLVM implementation details



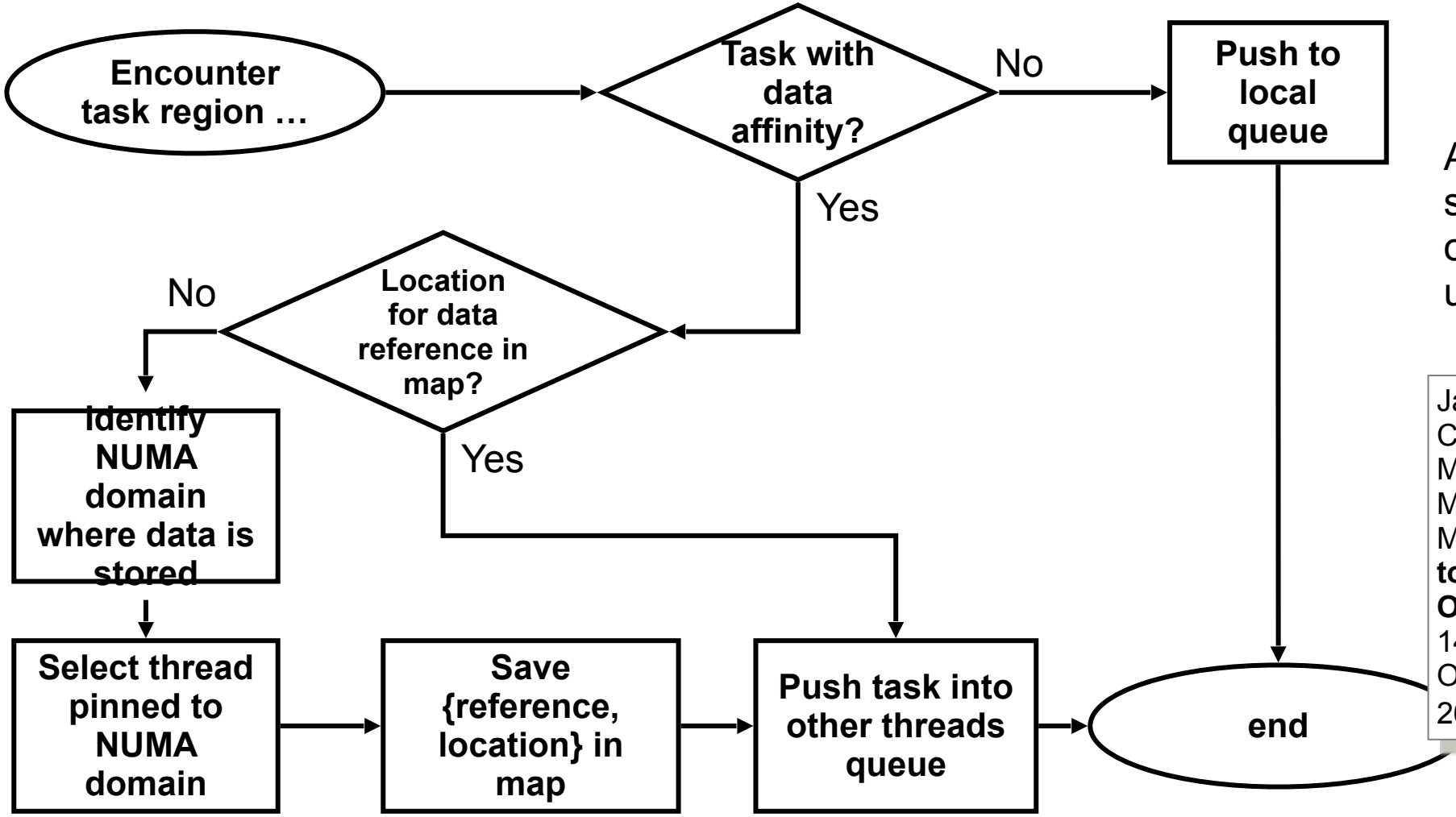

A map is introduced to store location information of data that was previously used

# Selected LLVM implementation details




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

# Selected LLVM implementation details

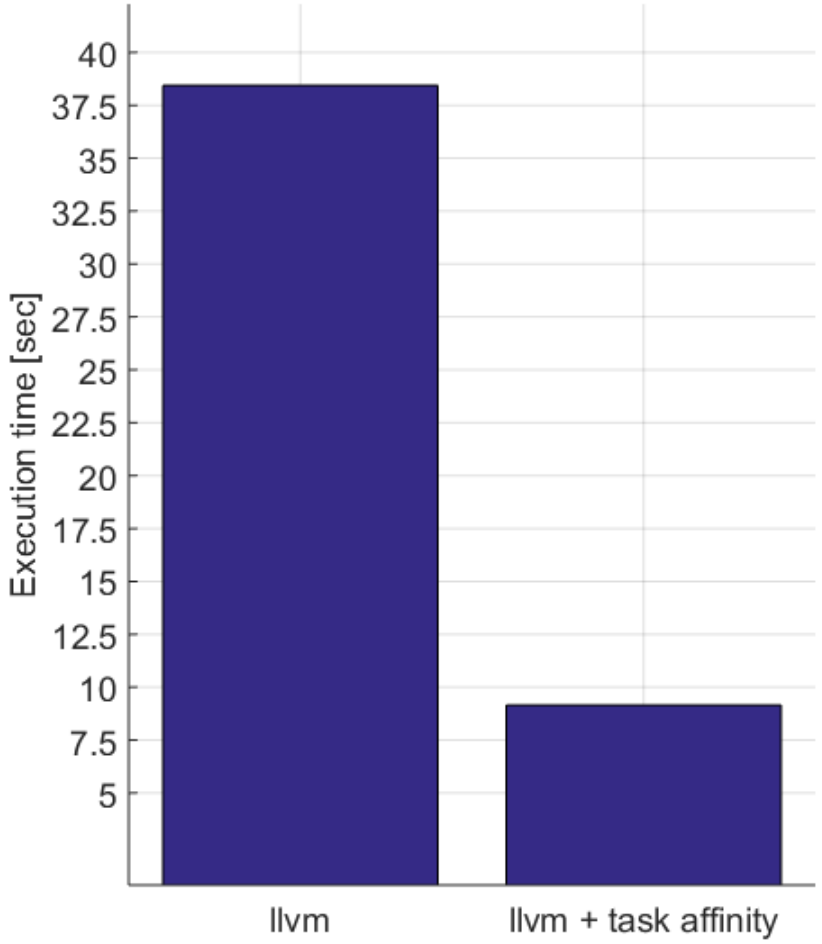



A map is introduced to store location information of data that was previously used

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.

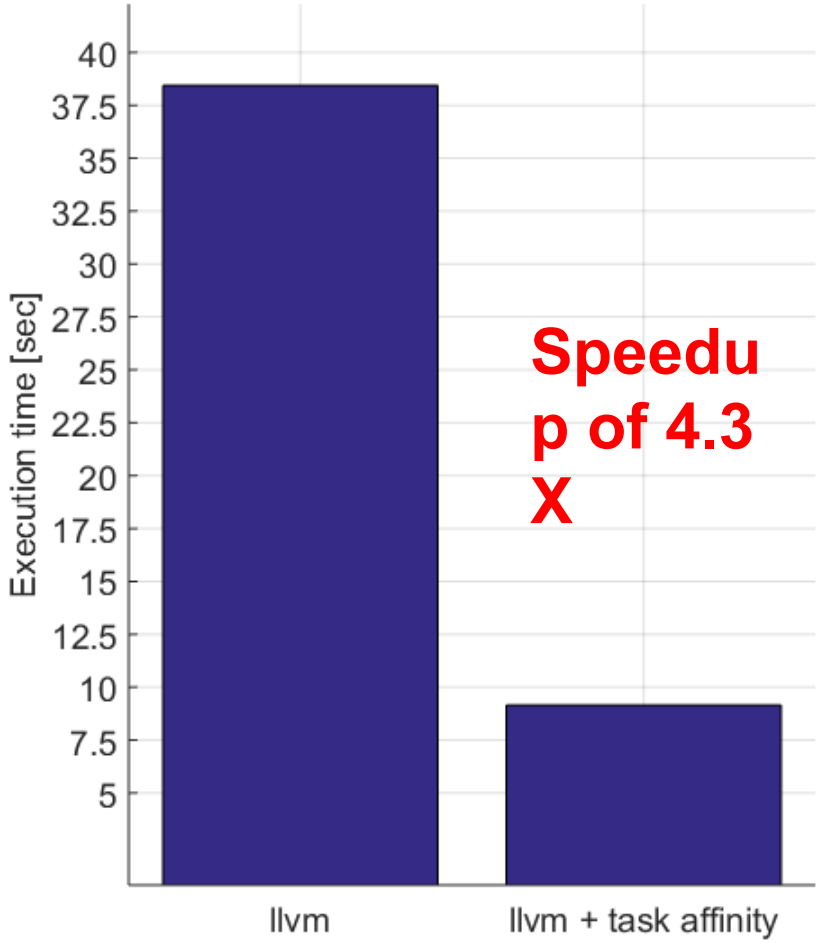
# Evaluation

## Program runtime
## Median of 10 runs



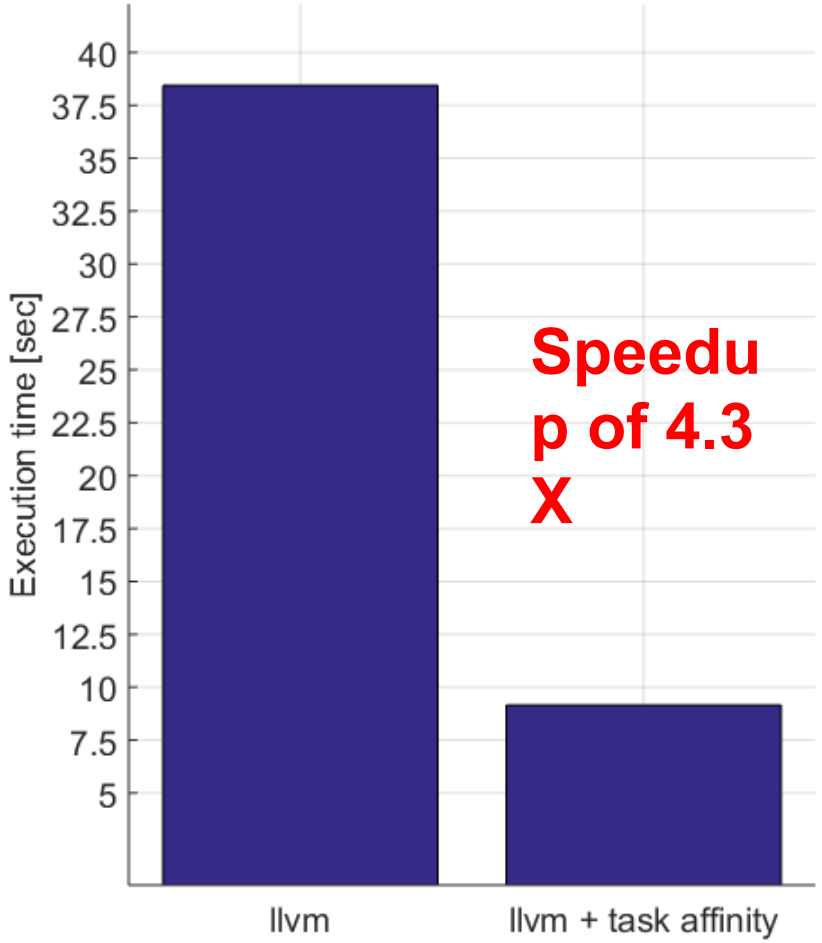
# Evaluation

**Program runtime**
**Median of 10 runs**



# Evaluation

**Program runtime**
**Median of 10 runs**

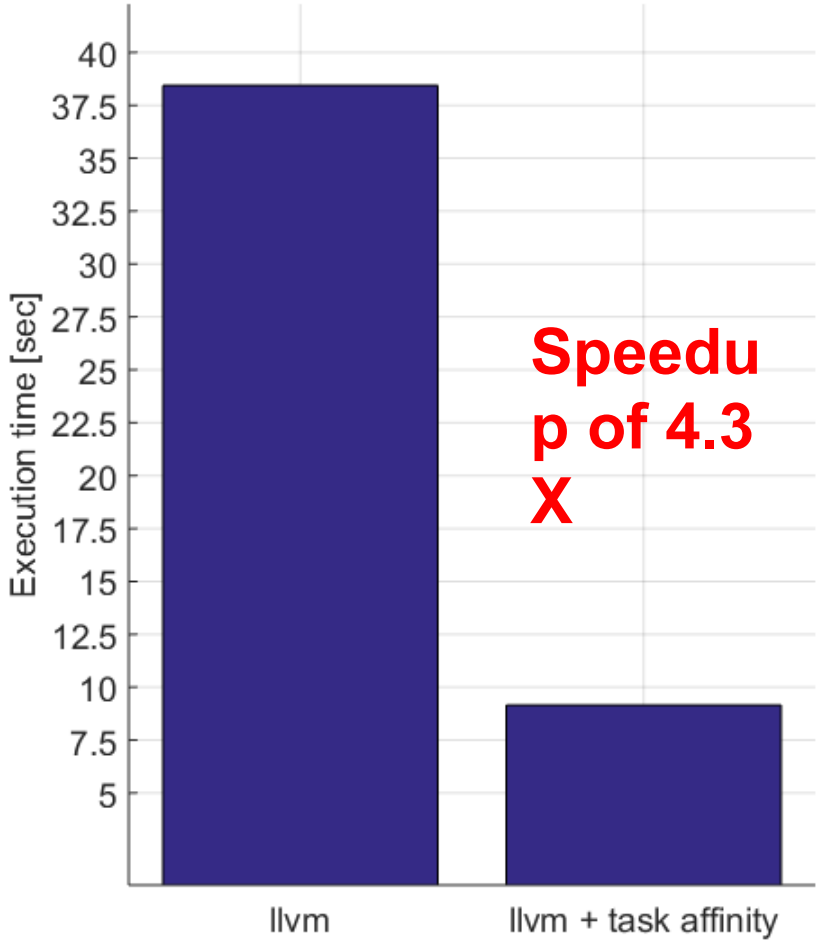



**Speedup of 4.3X**

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

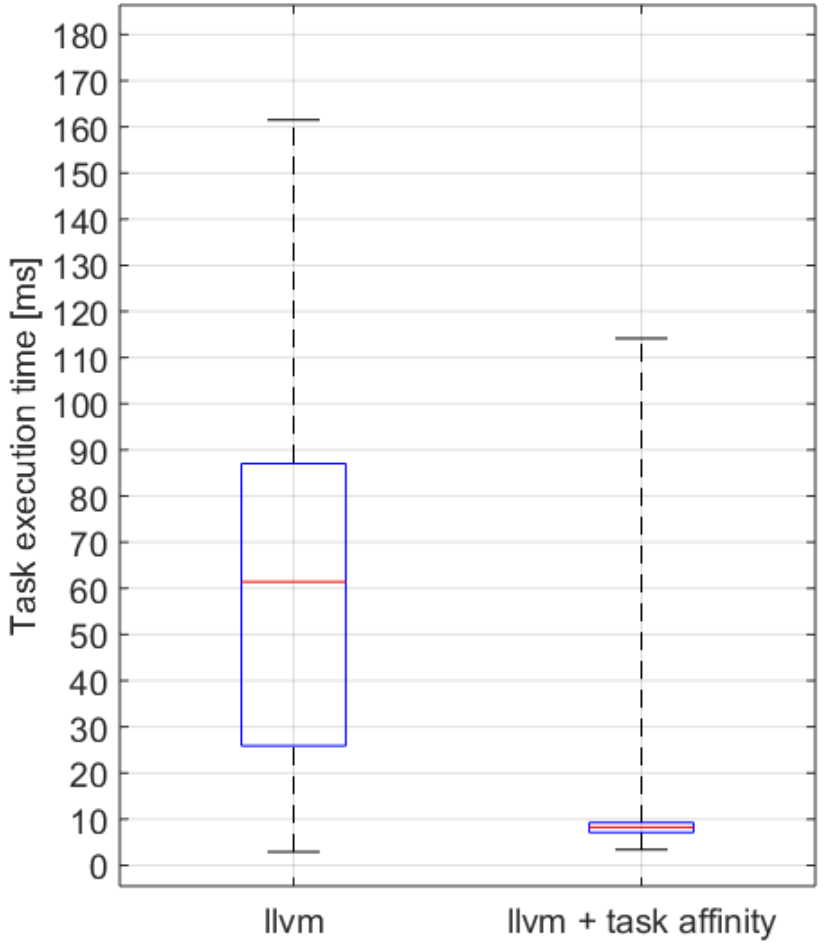
# Evaluation

## Program runtime
## Median of 10 runs




**Speedup of 4.3X**

## 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

# Programming OpenMP

***SIMD***

Christian Terboven

**Michael Klemm**

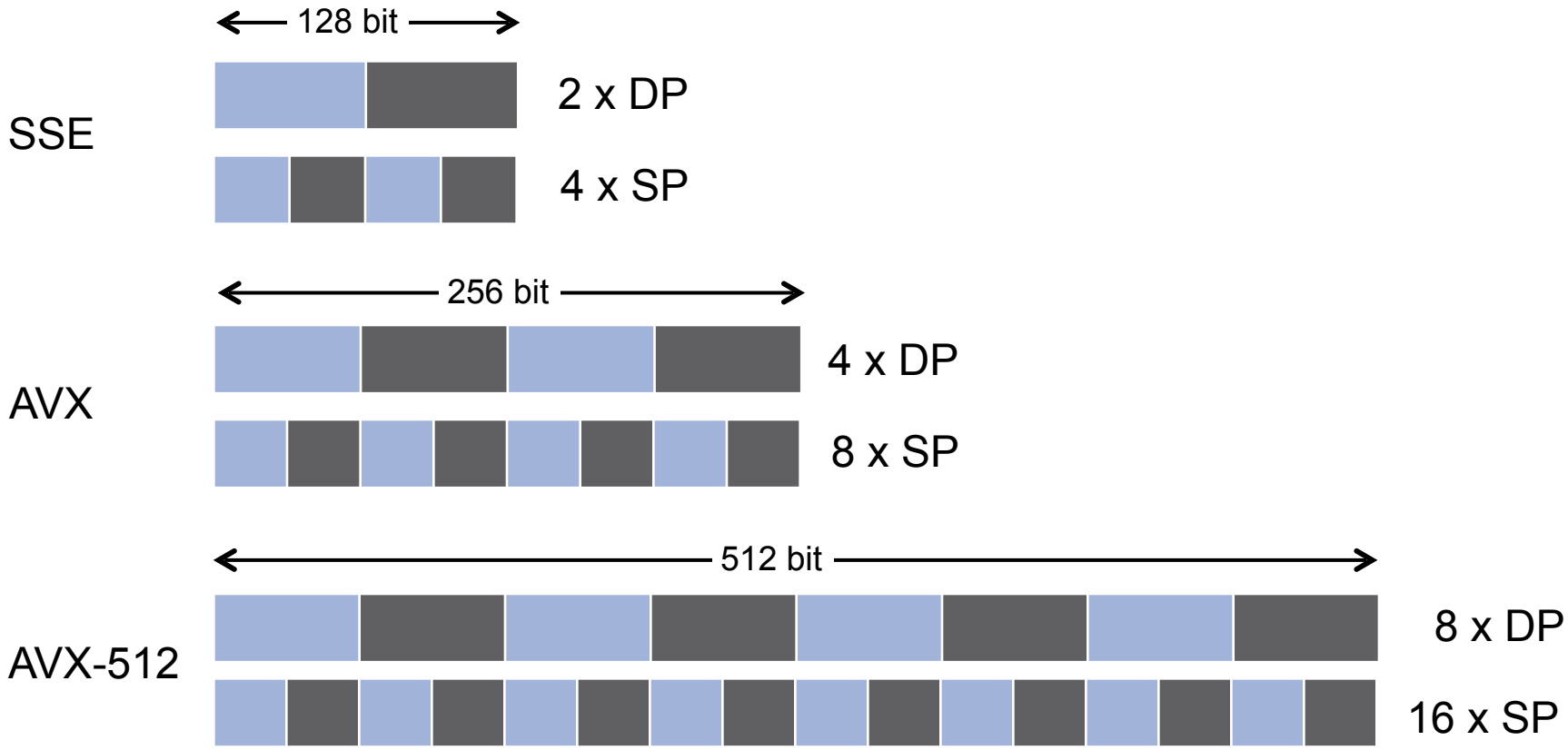
RWTH AACHEN UNIVERSITY

# Topics

- Exploiting SIMD parallelism with OpenMP
- Using SIMD directives with loops
- Creating SIMD functions

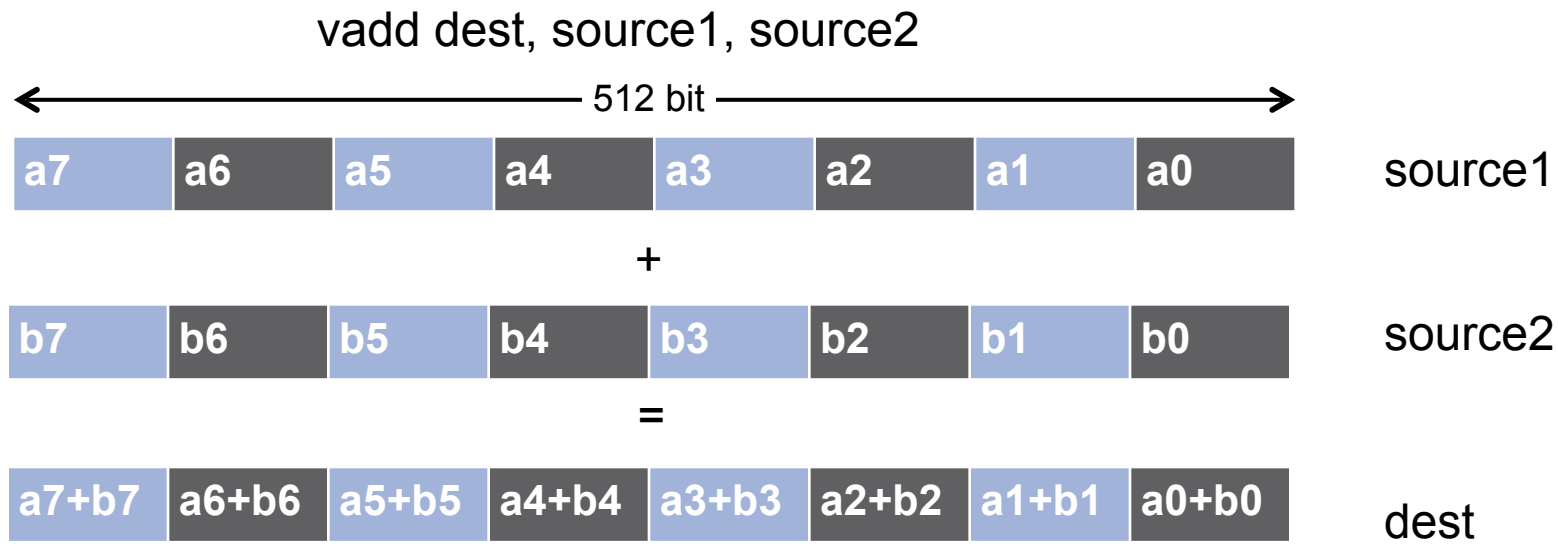
# SIMD on x86 Architectures

- Width of SIMD registers has been growing in the past:



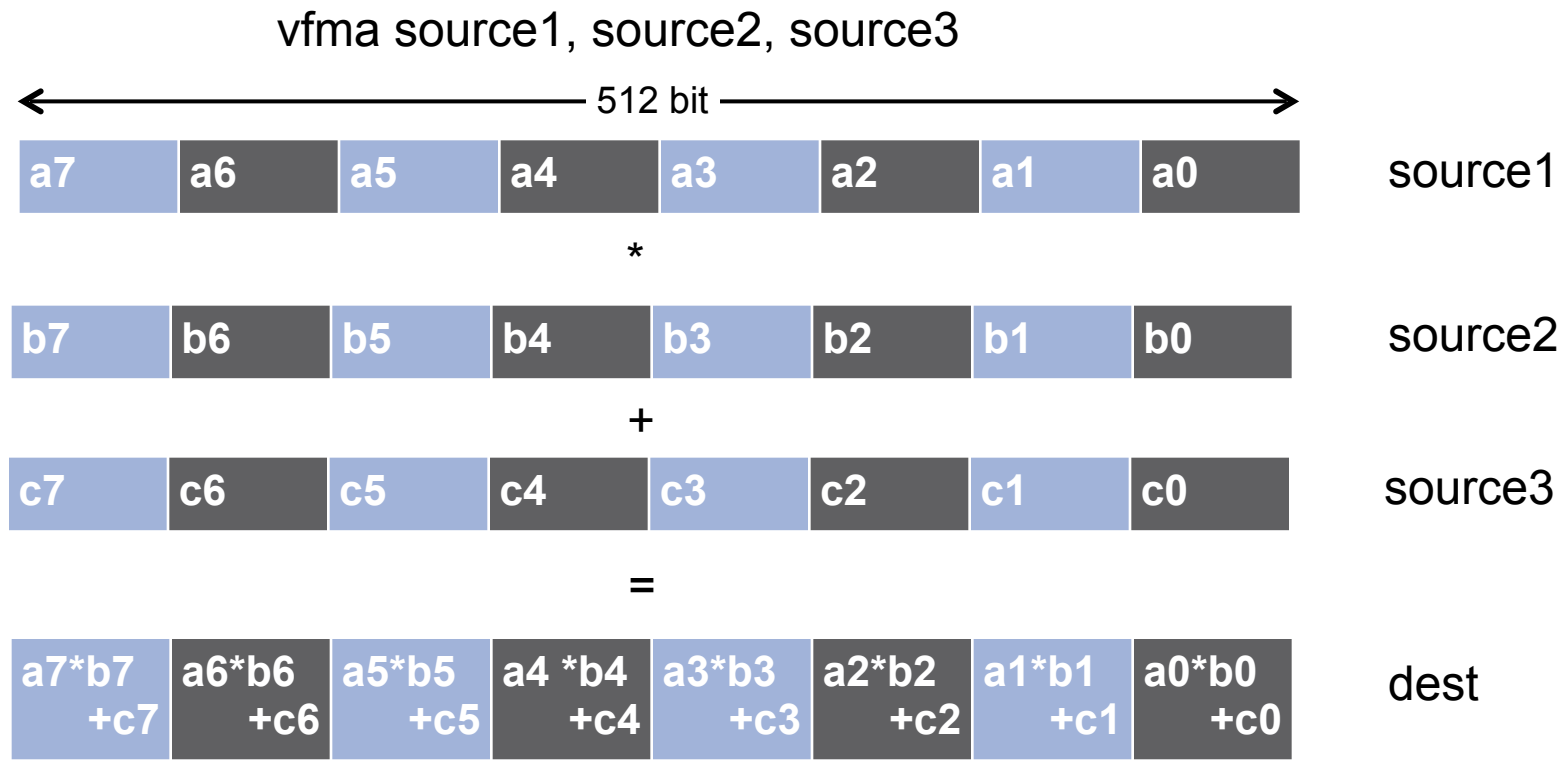
# More Powerful SIMD Units

- SIMD instructions become more powerful



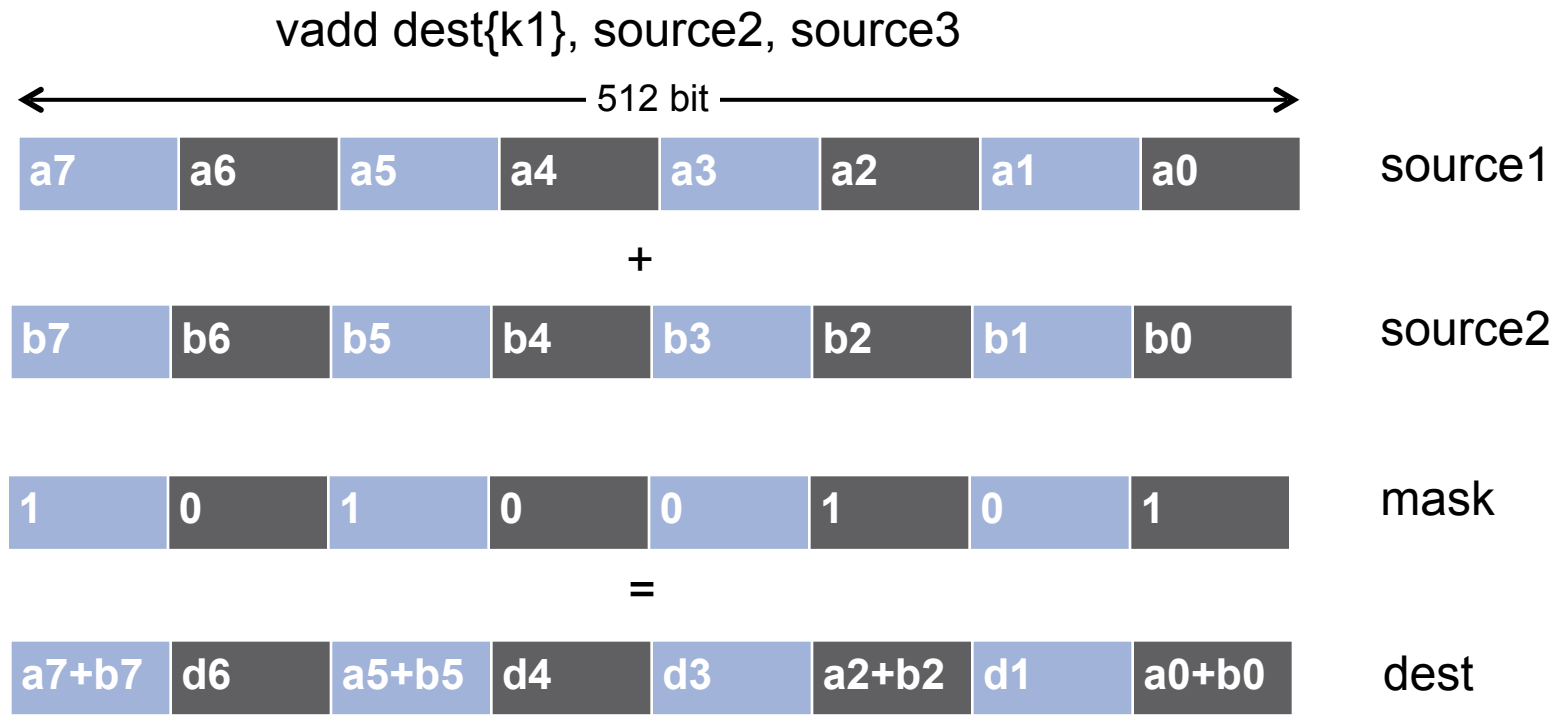
# More Powerful SIMD Units

- SIMD instructions become more powerful



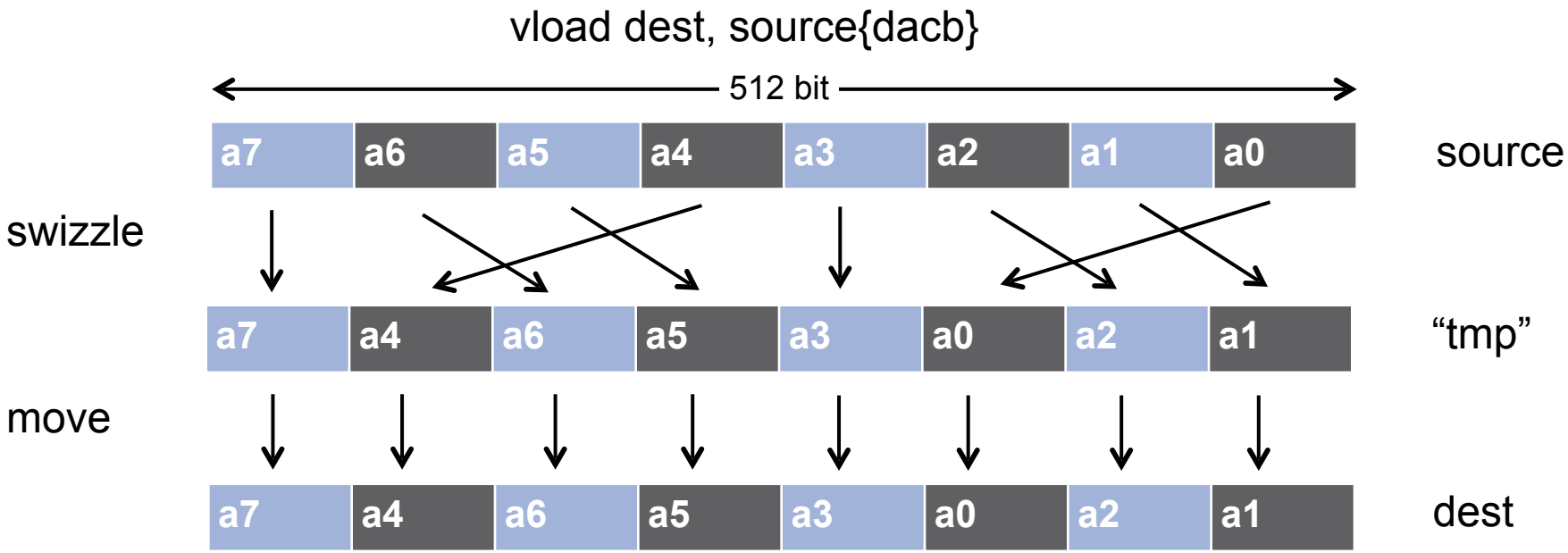
# More Powerful SIMD Units

- SIMD instructions become more powerful



# More Powerful SIMD Units

- SIMD instructions become more powerful



# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - →Usually, part of the general loop optimization passes

# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization

# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization
  - → Heuristics determine if SIMD execution might be beneficial

# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization
  - → Heuristics determine if SIMD execution might be beneficial
  - → If all goes well, the compiler will generate SIMD instructions

# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization
  - → Heuristics determine if SIMD execution might be beneficial
  - → If all goes well, the compiler will generate SIMD instructions

- Example: clang/LLVM
  - → -fvectorize
  - → -Rpass=loop-.*
  - → -mprefer-vector-width=<*width*>

GCC
-ftree-vectorize
-ftree-loop-vectorize
-fopt-info-vec-all

Intel Compiler
-vec (enabled w/ -O2)
-qopt-report=vec

# Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - → Usually, part of the general loop optimization passes
  - → Code analysis detects code properties that inhibit SIMD vectorization ?
  - → Heuristics determine if SIMD execution might be beneficial
  - → If all goes well, the compiler will generate SIMD instructions

- Example: clang/LLVM
  - → -fvectorize
  - → -Rpass=loop-.*
  - → -mprefer-vector-width=<*width*>

GCC
-ftree-vectorize
-ftree-loop-vectorize
-fopt-info-vec-all

Intel Compiler
-vec (enabled w/ -O2)
-qopt-report=vec

# Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - → Alignment
  - → Function calls in loop block
  - → Complex control flow / conditional branches
  - → Loop not “countable”
    - → e.g., upper bound not a runtime constant
  - → Mixed data types
  - → Non-unit stride between elements
  - → Loop body too complex (register pressure)
  - → Vectorization seems inefficient
- Many more … but less likely to occur

# Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - → Control-flow dependence
  - → Data dependence
  - → Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW

```
s1: a = 40

    b = 21

s2: c = a + 2
```

ANTI

```
    b = 40

s1: a = b + 1

s2: b = 21
```

# Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  → Loop-carried dependency

- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2)
{
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  → Simple trick: Can you reverse the loop w/o getting wrong results?

# Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  → Loop-carried dependency

- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2)
{
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

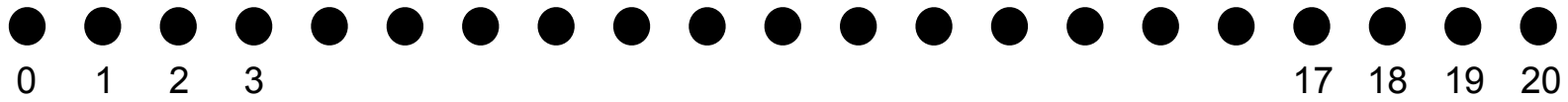
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- Some iterations of the loop have to complete before the next iteration can run
  → Simple trick: Can you reverse the loop w/o getting wrong results?

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

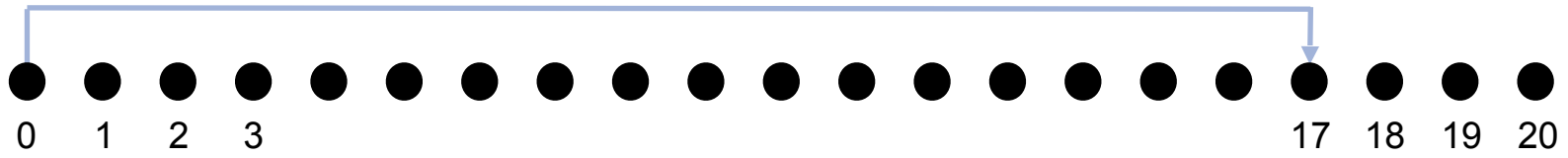
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

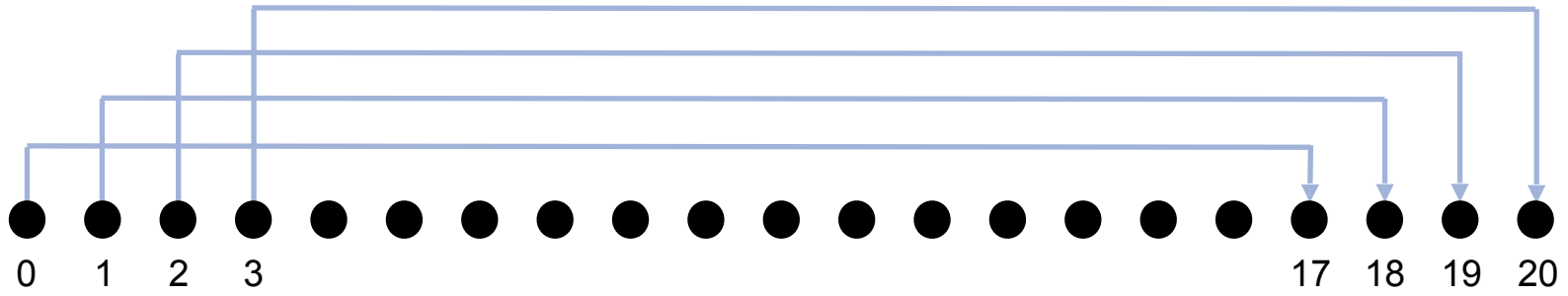
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

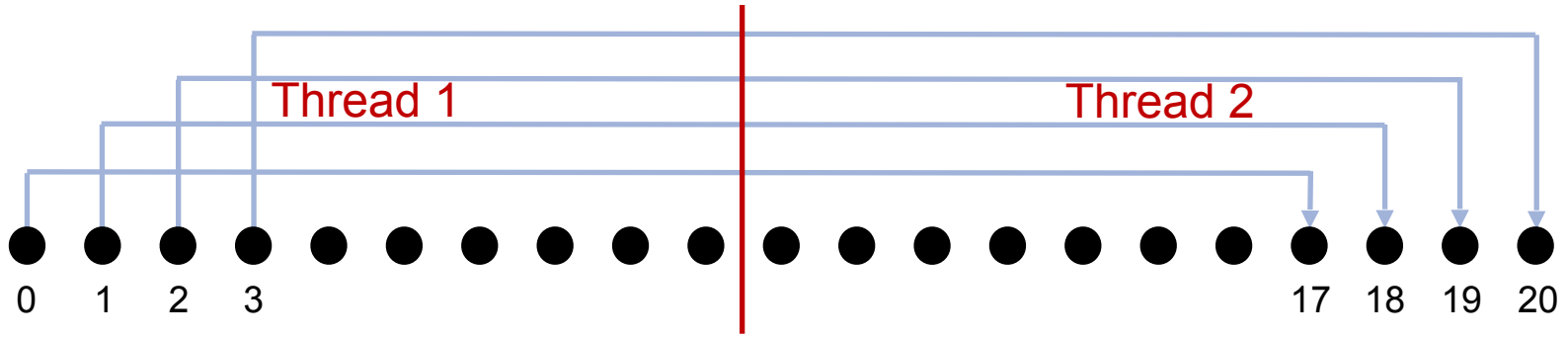
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

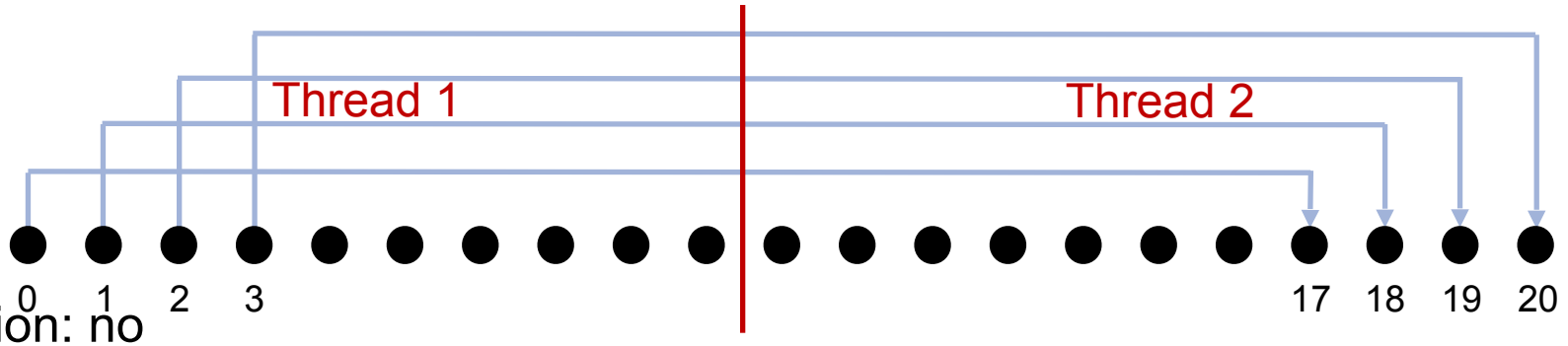
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



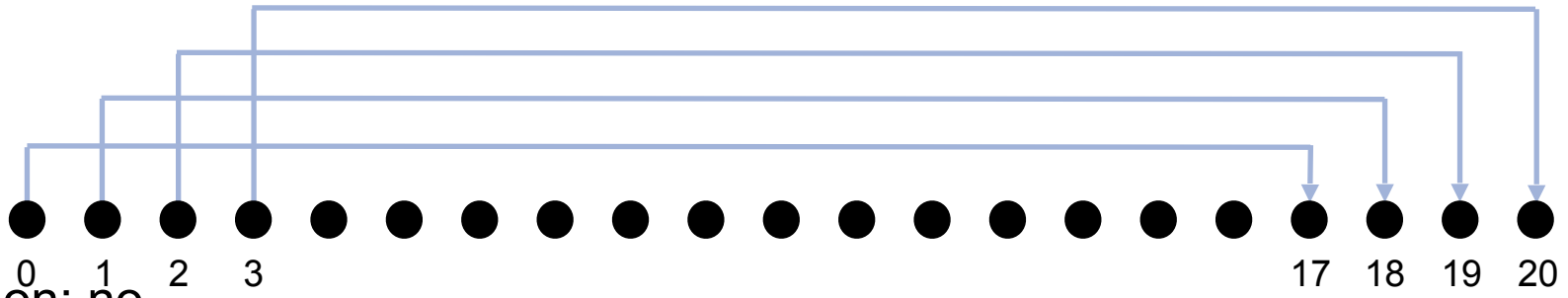

→ Parallelization: no

(except for very specific loop schedules)

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



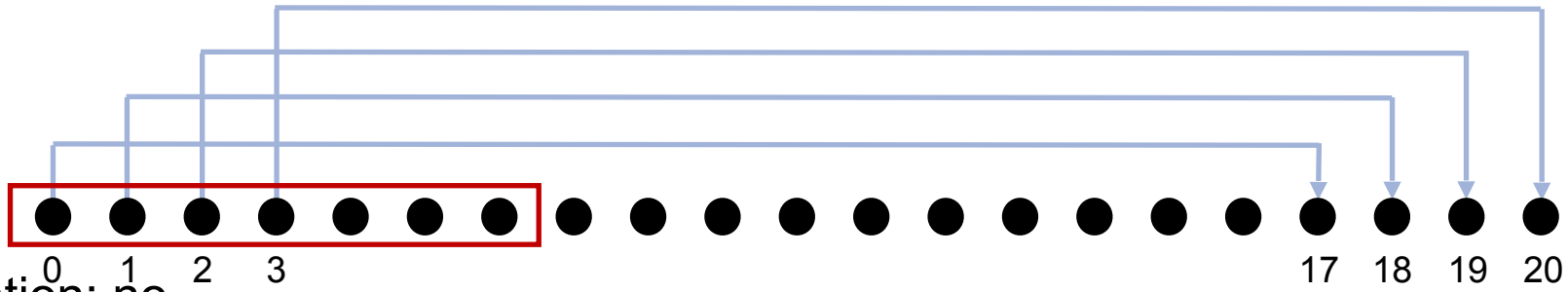

→ Parallelization: no

(except for very specific loop schedules)

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



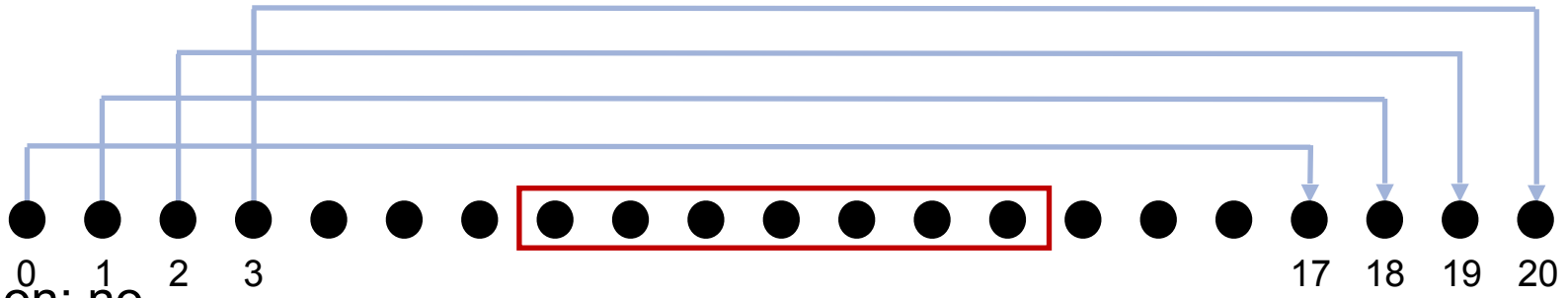

→ Parallelization: no

(except for very specific loop schedules)

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



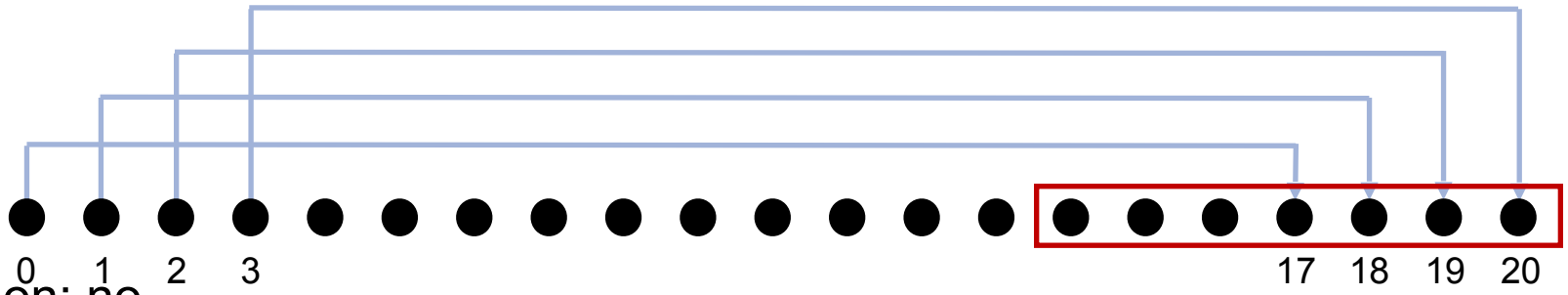

→ Parallelization: no

(except for very specific loop schedules)

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



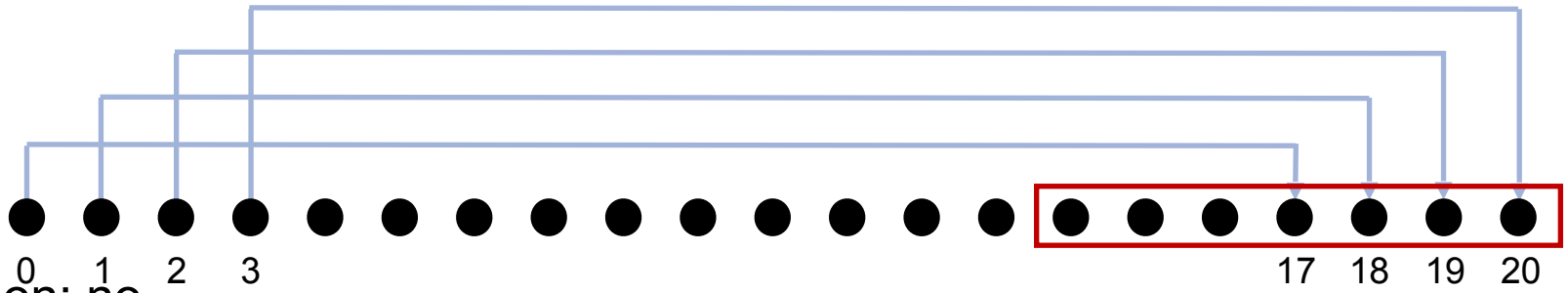

→ Parallelization: no

(except for very specific loop schedules)

# Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```




→ Parallelization: no

(except for very specific loop schedules)

→ Vectorization: yes

(iff vector length is shorter than any distance of any dependency)

# In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - →Programming models (e.g., Intel® Cilk Plus)
  - →Compiler pragmas (e.g., `#pragma vector`)
  - →Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

# In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - →Programming models (e.g., Intel® Cilk Plus)
  - →Compiler pragmas (e.g., `#pragma vector`)
  - →Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

# SIMD Loop Construct

- Vectorize a loop nest
  - →Cut loop into chunks that fit a SIMD vector register
  - →No parallelization of the loop body

- Syntax (C/C++)

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

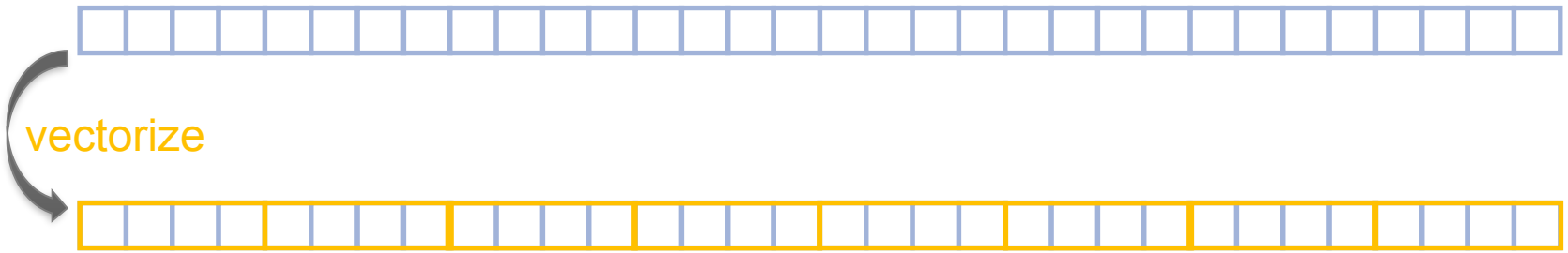
- Syntax (Fortran)

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

# Example

# Example

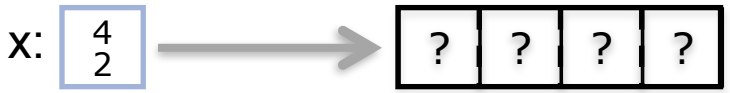
```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# Data Sharing Clauses

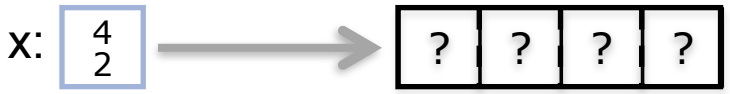
# Data Sharing Clauses

- `private(var-list)`:
  Uninitialized vectors for variables in *var-list*

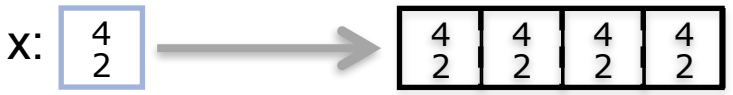


# Data Sharing Clauses

- `private(var-list):`
  Uninitialized vectors for variables in *var-list*

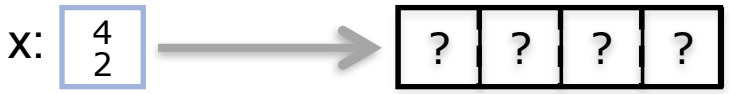



- `firstprivate(var-list):`
  Initialized vectors for variables in *var-list*

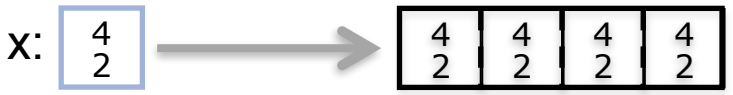


# Data Sharing Clauses

- `private(var-list):`
  Uninitialized vectors for variables in *var-list*




- `firstprivate(var-list):`
  Initialized vectors for variables in *var-list*




- `reduction(op:var-list):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct

# SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length

# SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$

# SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture

# SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture
- `collapse (n)`

# SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - → Distribute a loop's iteration space across a thread team
  - → Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

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

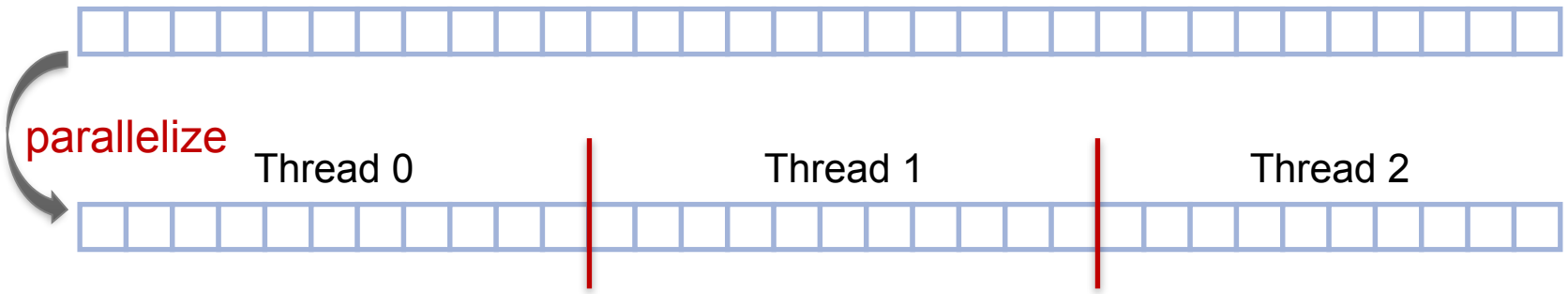
- Syntax (Fortran)

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

# Example

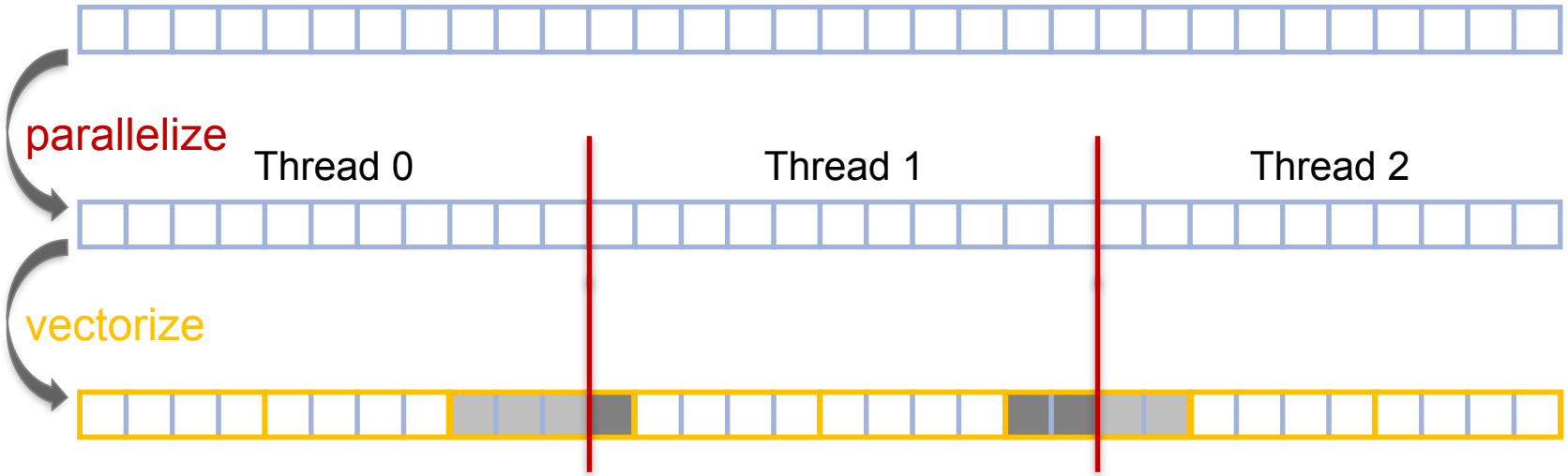
# Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



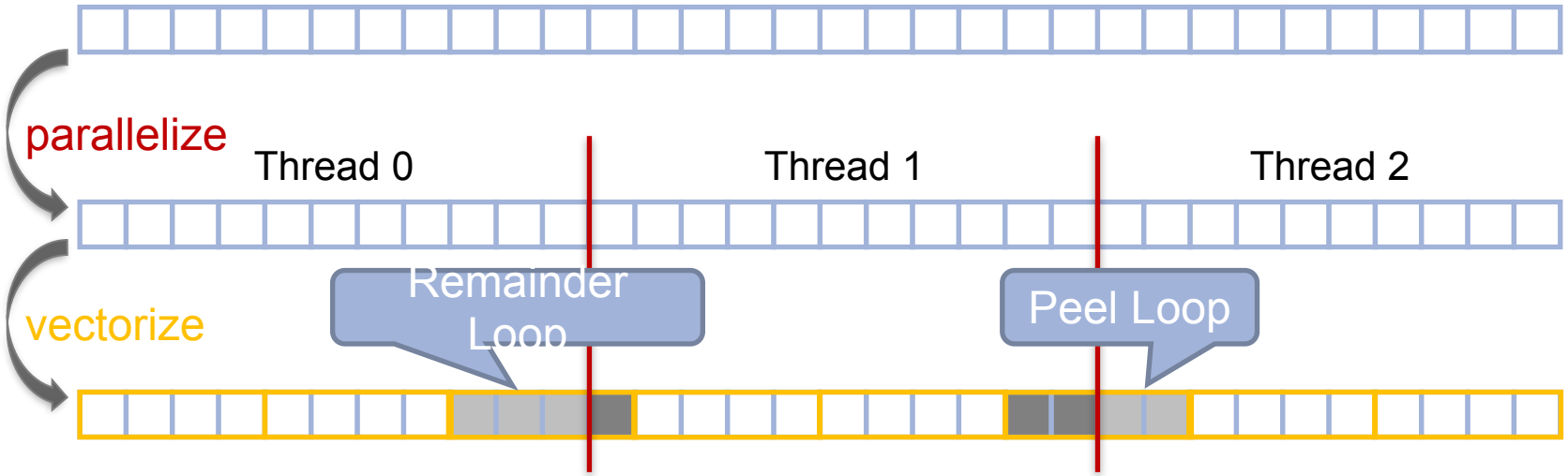
# Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# Be Careful What You Wish For…

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance

# Be Careful What You Wish For…

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance

# Be Careful What You Wish For…

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance
- In the above example …
  - → and AVX2, the code will only execute the remainder loop!
  - → and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

# OpenMP 4.5 Simplifies SIMD Chunks

- Chooses chunk sizes that are multiples of the SIMD length
  - →First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
  - →Remainder loops are not triggered
  - →Likely better performance

# OpenMP 4.5 Simplifies SIMD Chunks

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(simd: static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- Chooses chunk sizes that are multiples of the SIMD length
  - → First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
  - → Remainder loops are not triggered
  - → Likely better performance

# SIMD Function Vectorization

# SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

# SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]

[#pragma omp declare simd [clause[[,] clause],…]]

[…]

function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

# SIMD Function Vectorization

# SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

# SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):
    vminps %zmm1, %zmm0, %zmm0
    ret
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):
    vsubps %zmm0, %zmm1, %zmm2
    vmulps %zmm2, %zmm2, %zmm0
    ret
```

```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

# SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length

# SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length
- `uniform (`*`argument-list`*`)`
  - → argument has a constant value between the iterations of a given loop

# SIMD Function Vectorization

- `simdlen (length)`
  - → generate function to support a given vector length
- `uniform (argument-list)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement

# SIMD Function Vectorization

- `simdlen (length)`
  → generate function to support a given vector length
- `uniform (argument-list)`
  → argument has a constant value between the iterations of a given loop
- `inbranch`
  → function always called from inside an if statement
- `notinbranch`
  → function never called from inside an if statement
- `linear (argument-list[:linear-step])`
- `aligned (argument-list[:alignment])`

# inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

# inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

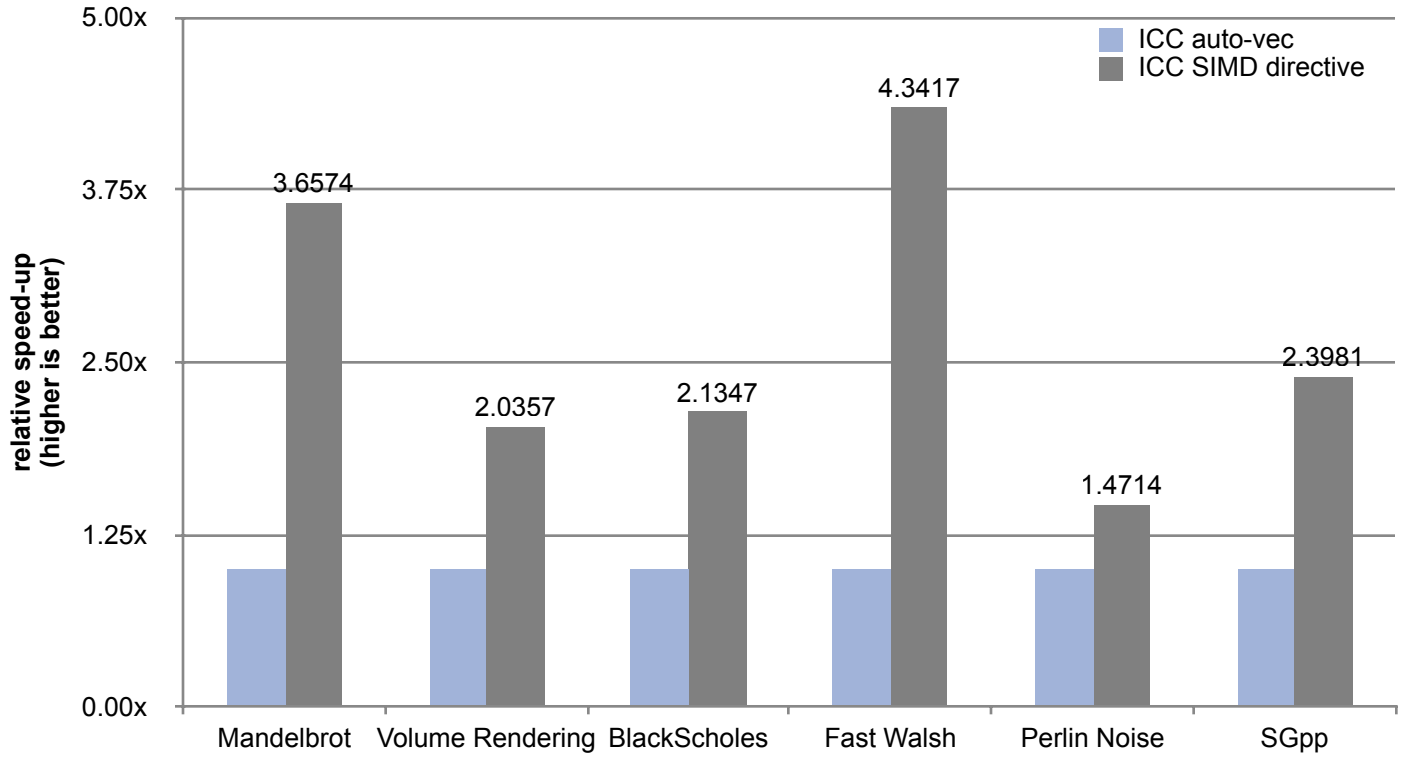
# inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

# SIMD Constructs & Performance




M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

# Programming OpenMP

***Hands-on Exercises***

**Christian Terboven**
Michael Klemm
Yun (Helen) He

# Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - → Download: `git clone https://github.com/NERSC/openmp-series-2024`
  - → Subfolder: `Session-3-NUMA_SIMD/exercises`, with instructions in `Exercises_OMP_2024.pdf`
  - → Build: `make` (or follow README files)
  - → You can then find the compiled executable to run with sample Slurm commands
  - → We use the GCC compiler mostly

- Each hands-on exercise has a folder “solution”
  - → It shows the OpenMP directive that we have added
  - → You can use it to cheat ☺, or to check if you came up with the same solution

# Exercises: Overview

| Exercise no. | Exercise name | OpenMP Topic | Day / Order (proposal) |
|---|---|---|---|
| 1 | PI | Apply OpenMP SIMD | Third day |
| 2 | xthi | Review for NUMA | Third day |
| 3 | Stream | Optimize / review for NUMA | Third day |
| 4 | Jacobi | Optimize / review for NUMA | Third day |