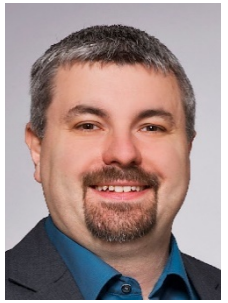
Programming OpenMP

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

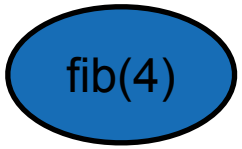
Fibonacci illustrated

```
 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.}
```

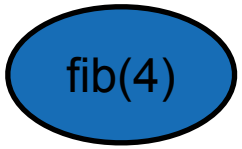
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

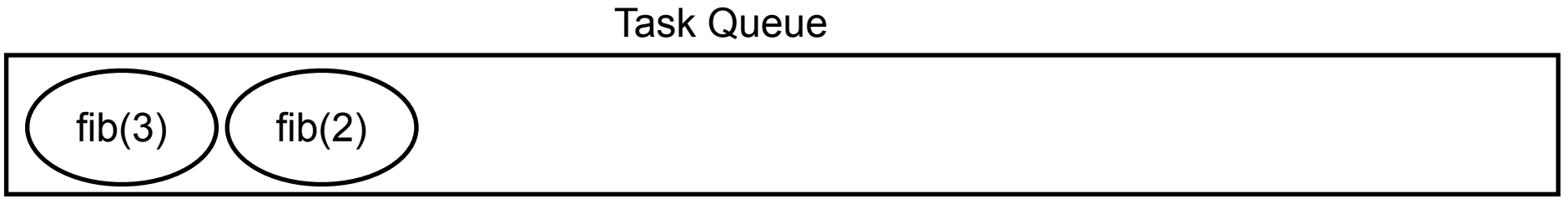
- **T1 enters fib(4)**



Task Queue

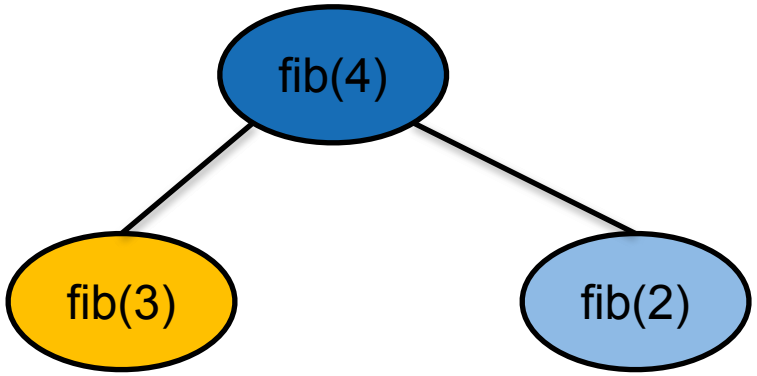
- T1 enters fib(4)
- **T1 creates tasks for fib(3) and fib(2)**





- T1 enters fib(4)

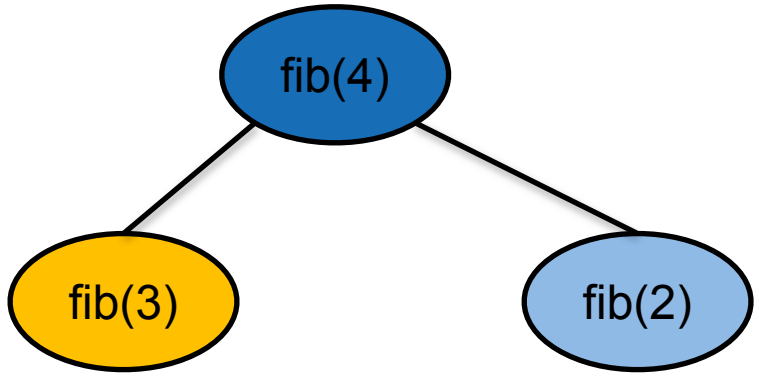
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**

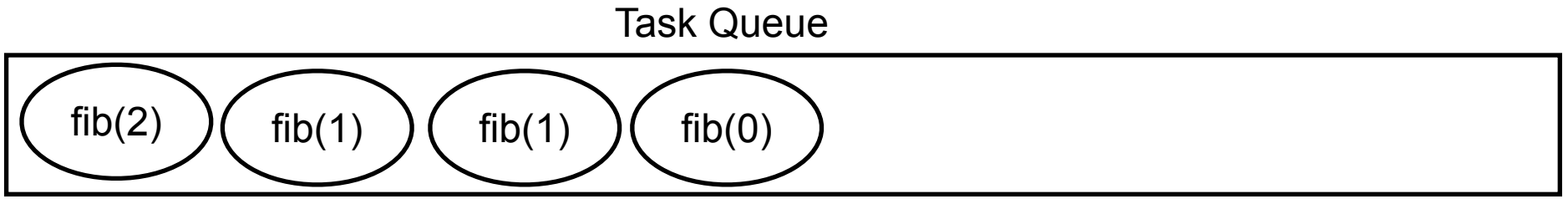


Task Queue

- T1 enters fib(4)

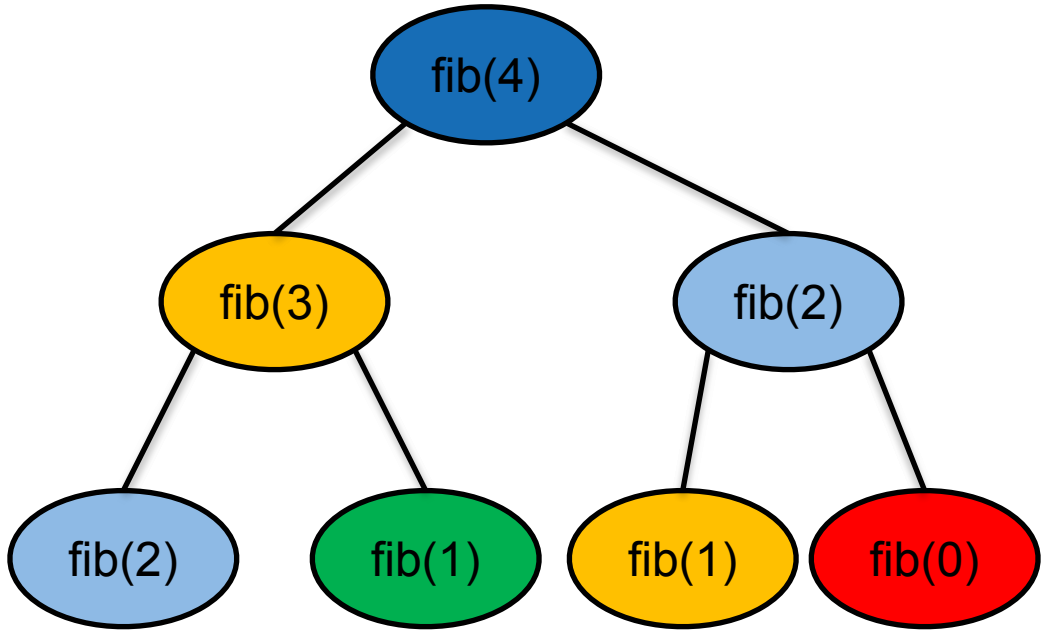
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks





- T1 enters fib(4)

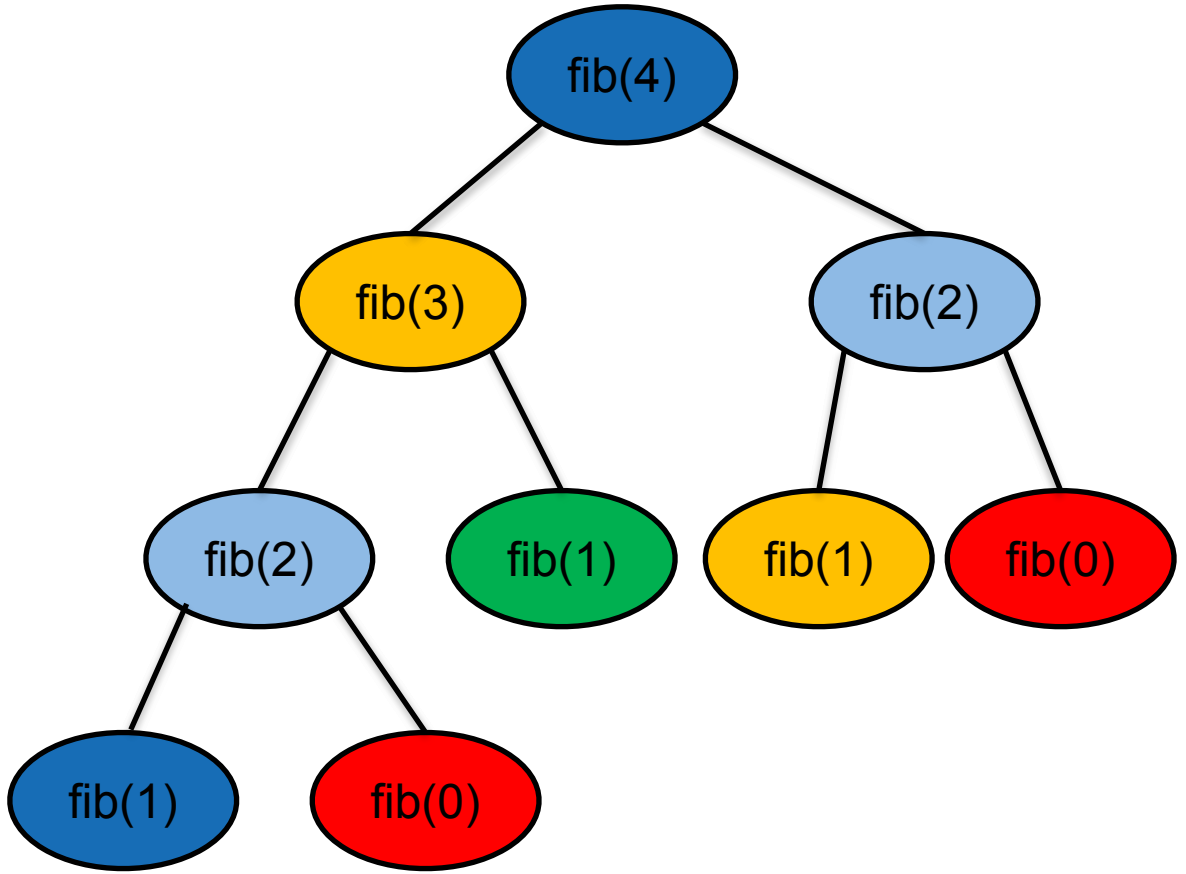
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



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

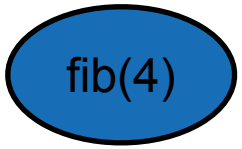
Fibonacci illustrated

```
 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.}
```

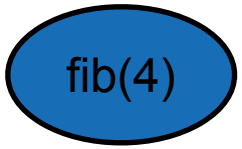
- Only one Task / Thread enters fib() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would get lost

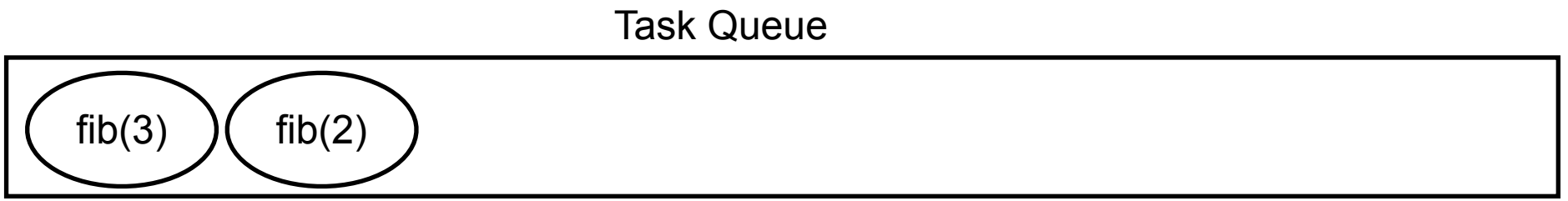
- **T1 enters fib(4)**



Task Queue

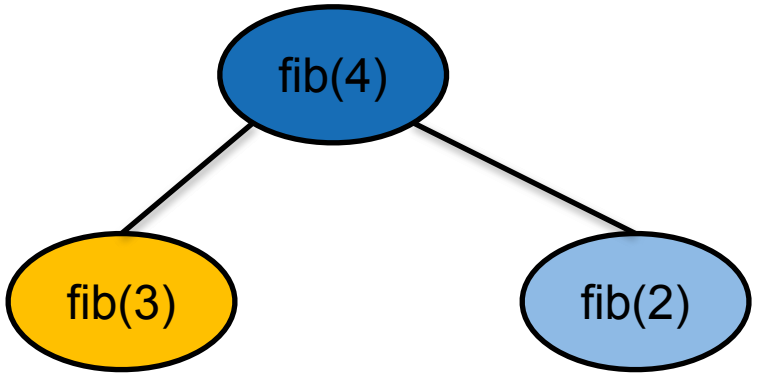
- T1 enters fib(4)
- **T1 creates tasks for fib(3) and fib(2)**





- T1 enters fib(4)

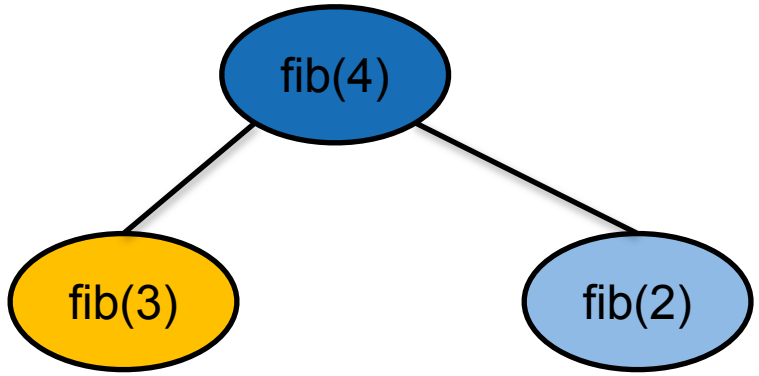
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**

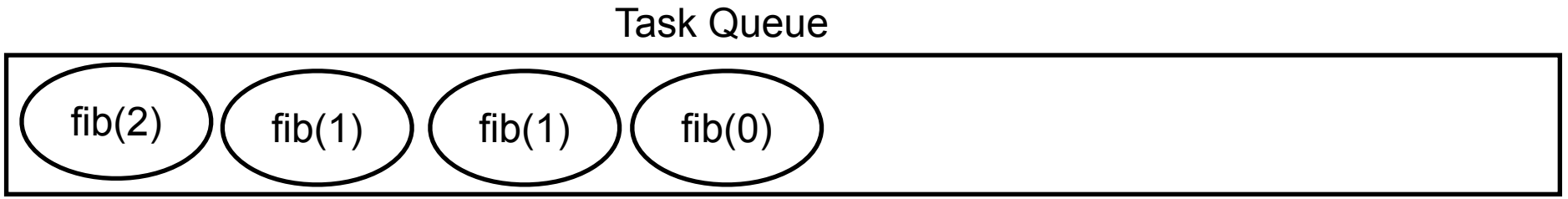


Task Queue

- T1 enters fib(4)

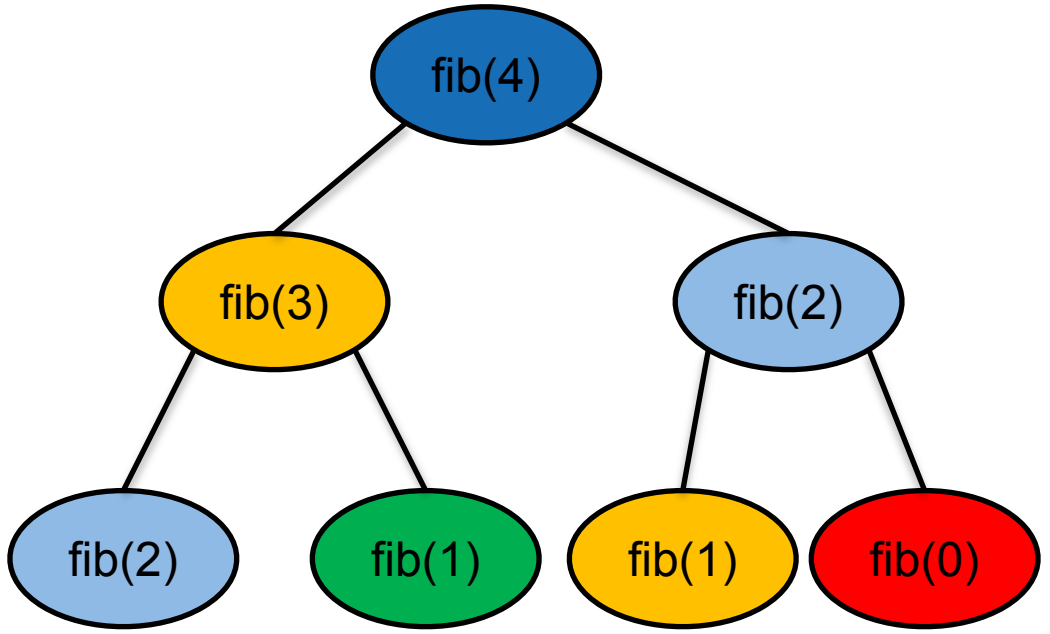
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks





- T1 enters fib(4)

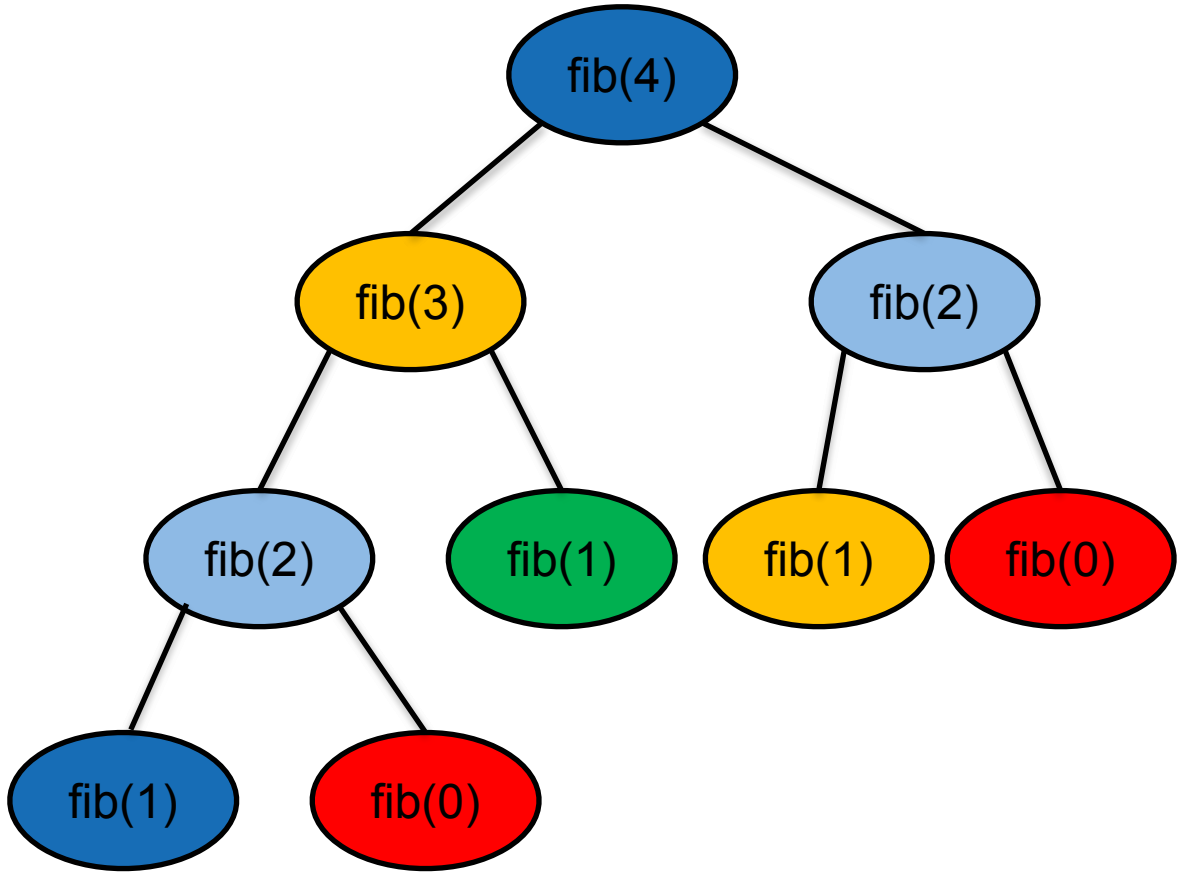
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



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

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

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`

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

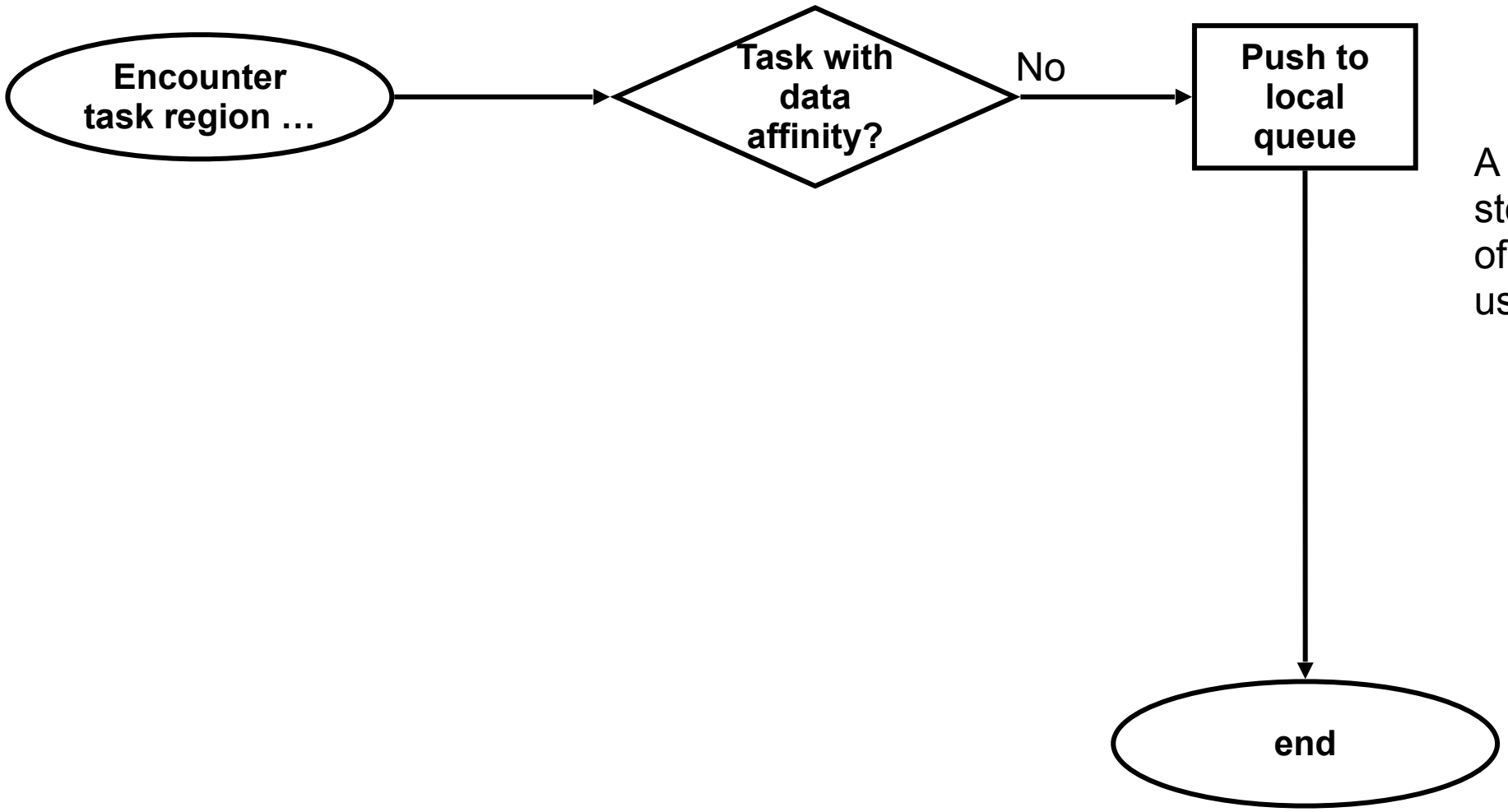
- Excerpt from task-parallel STREAM

```
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`)

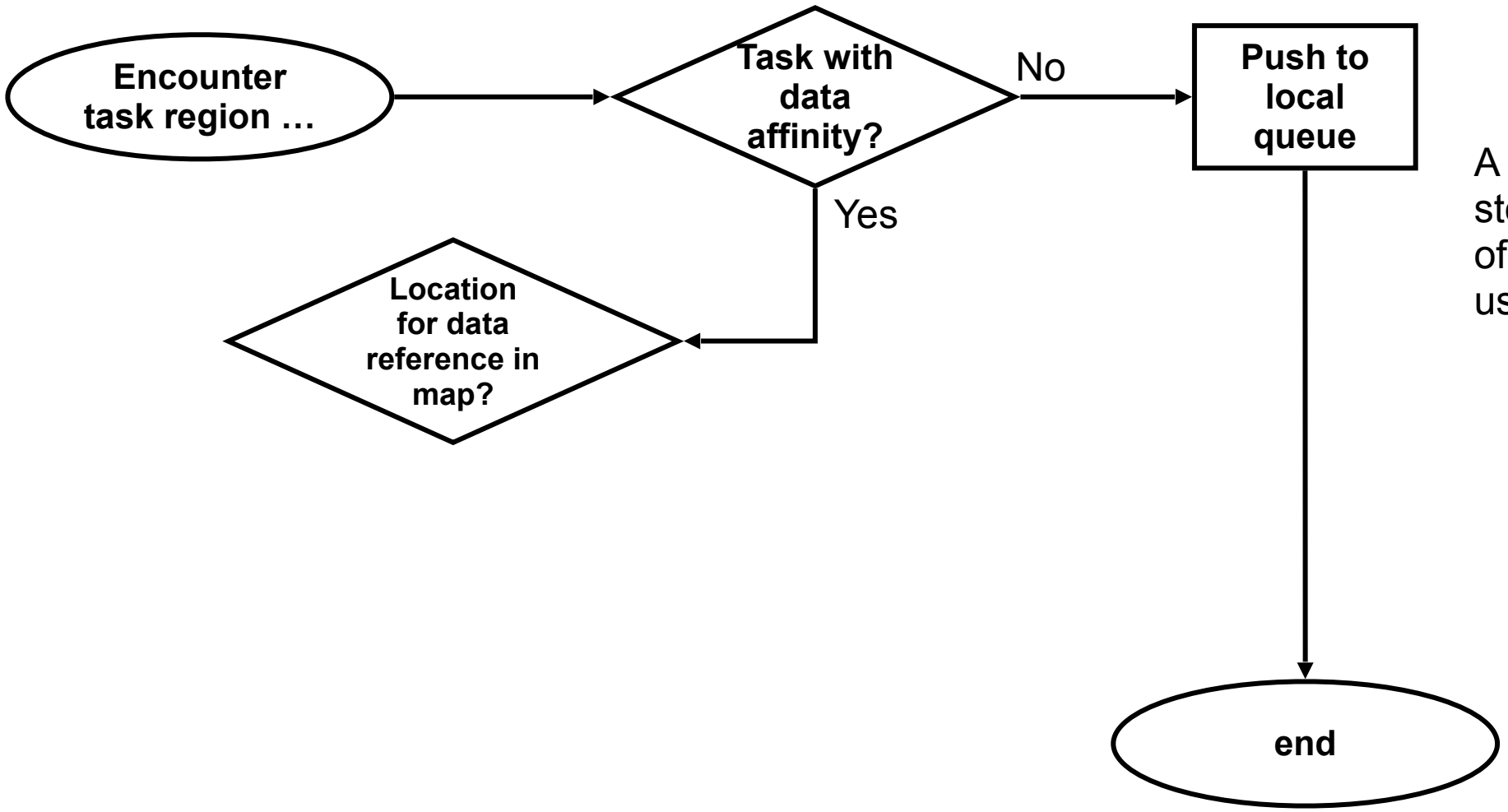
→Assumption: initialization and computation have same blocking and same affinity

Selected LLVM implementation details



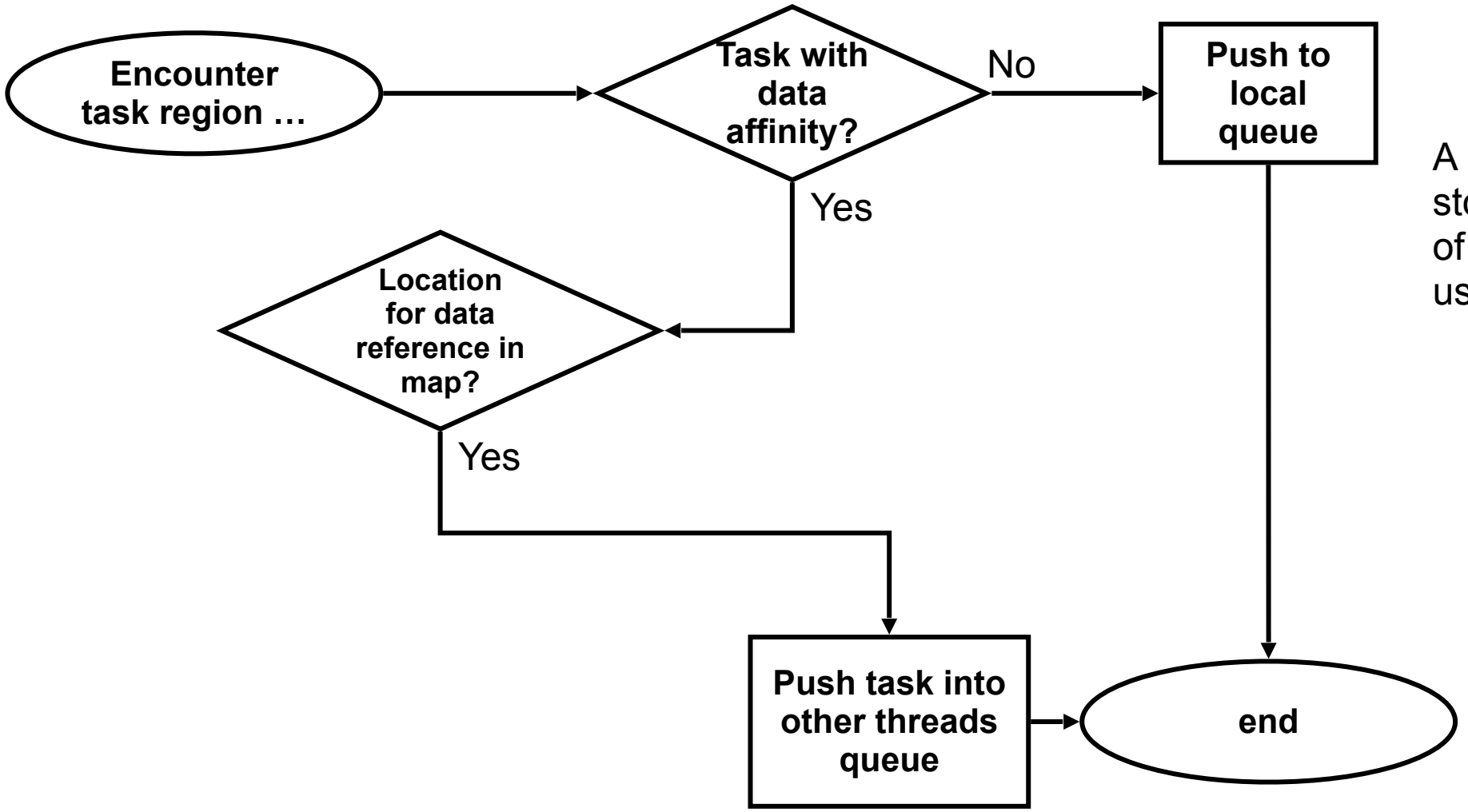
A map is introduced to store location information of data that was previously used

Selected LLVM implementation details



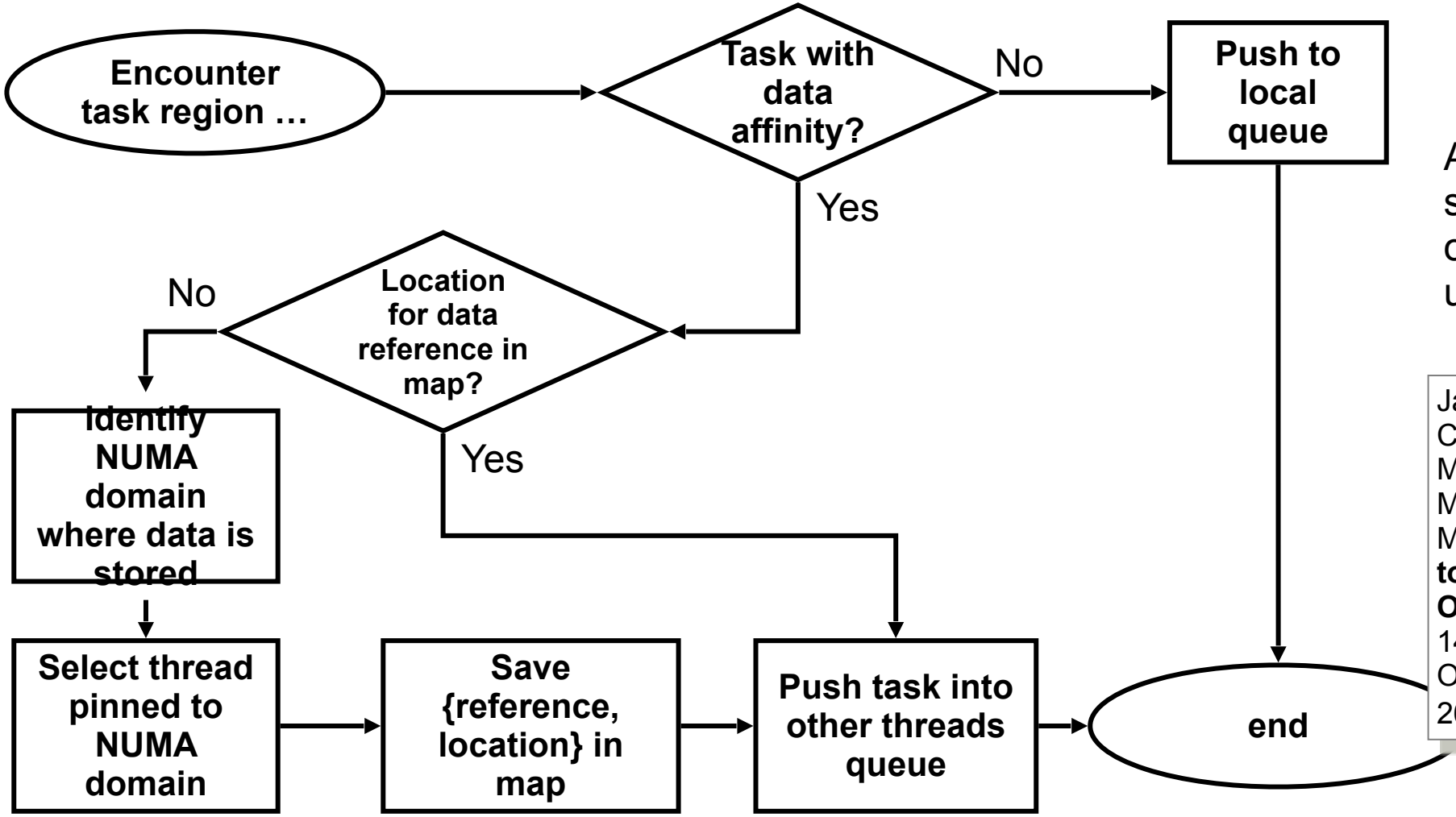
A map is introduced to store location information of data that was previously used

Selected LLVM implementation details



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

Selected LLVM implementation details

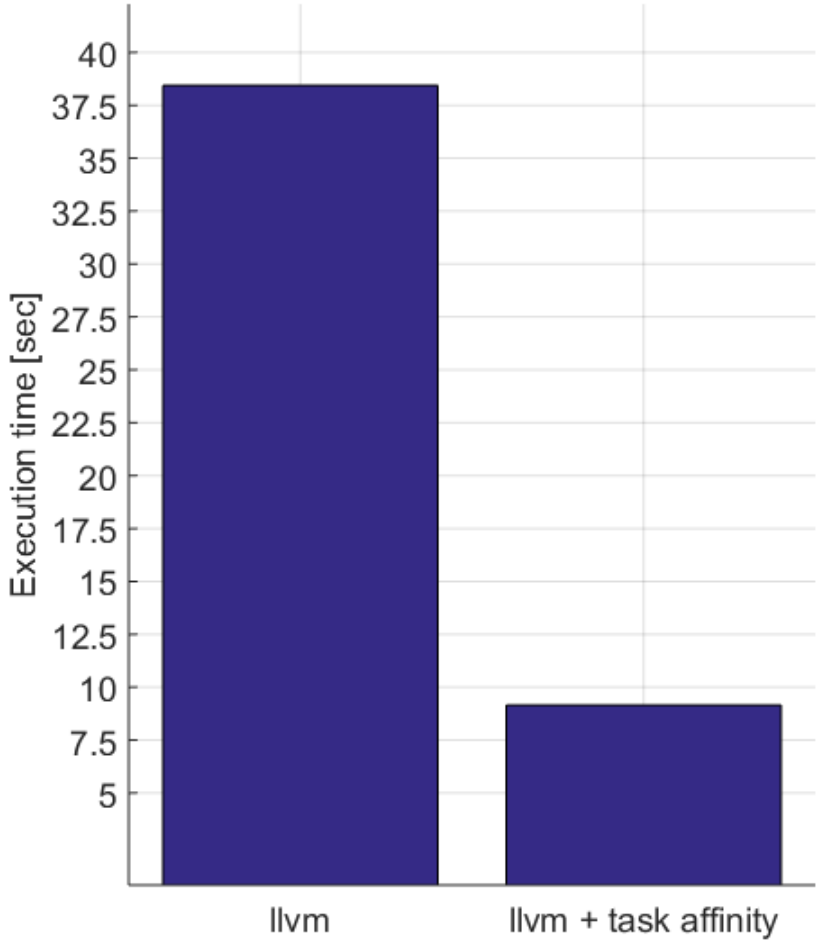


A map is introduced to store location information of data that was previously used

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.

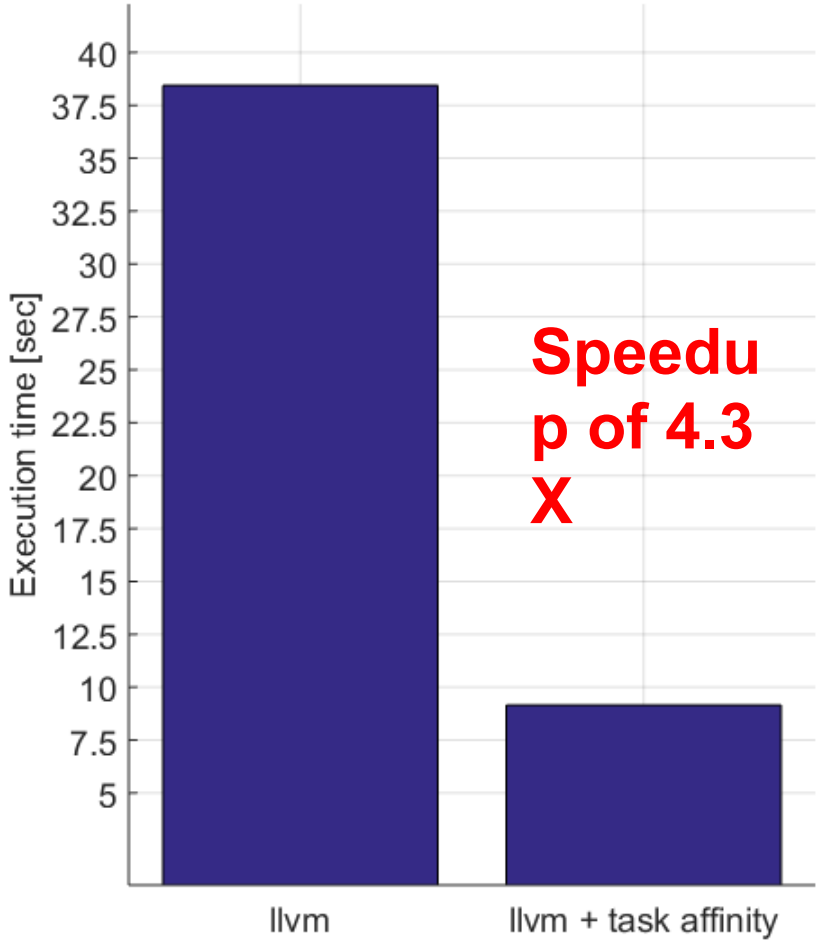
Evaluation

Program runtime
Median of 10 runs



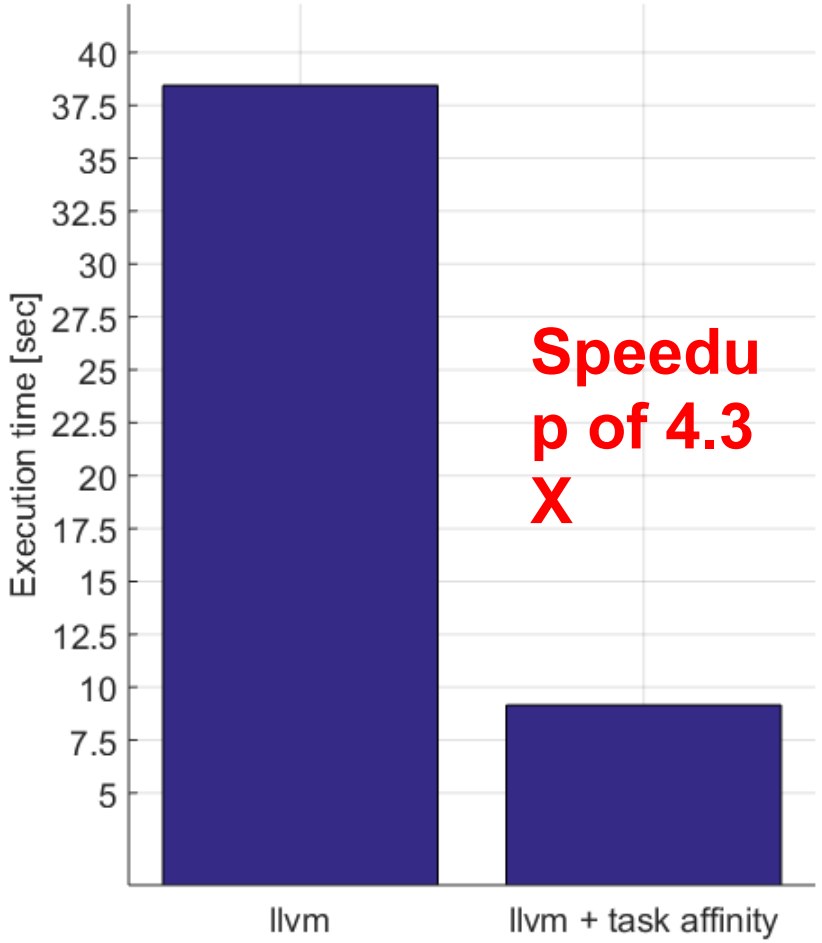
Evaluation

Program runtime
Median of 10 runs



Evaluation

Program runtime
Median of 10 runs

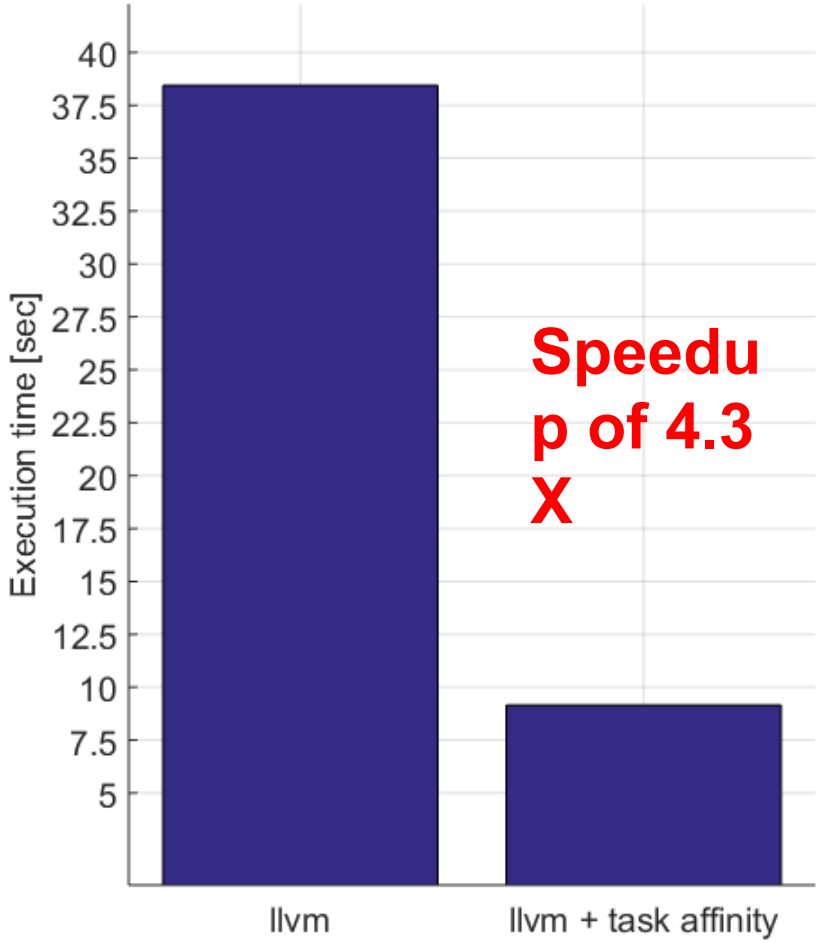


Speedup of 4.3X

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

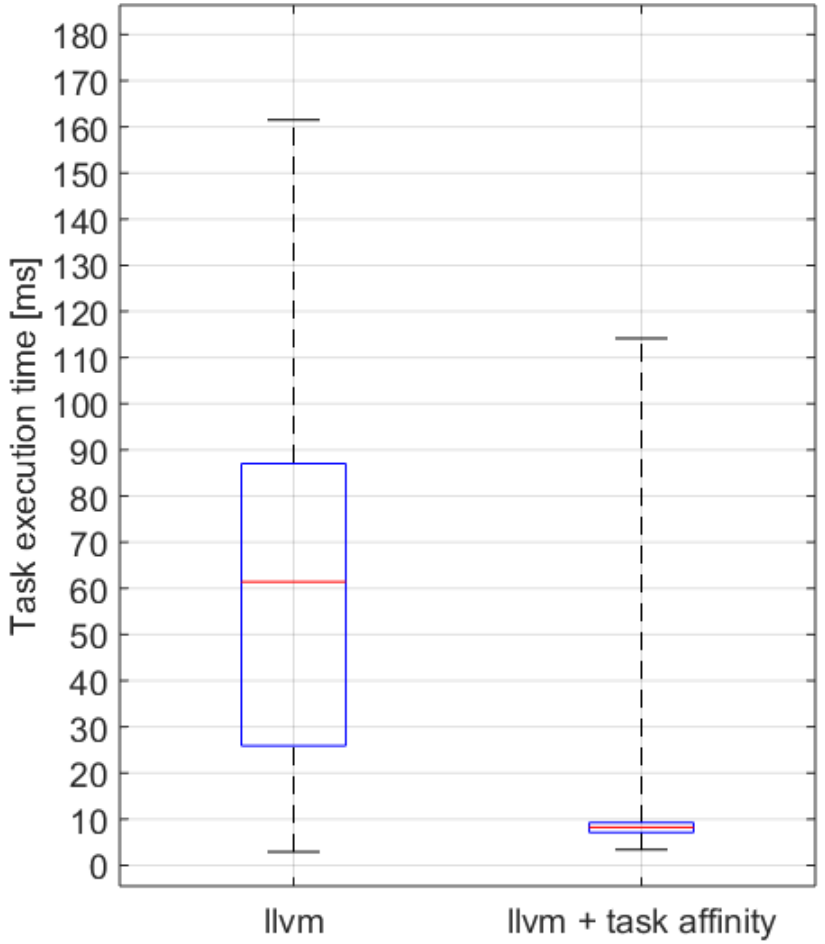
Evaluation

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

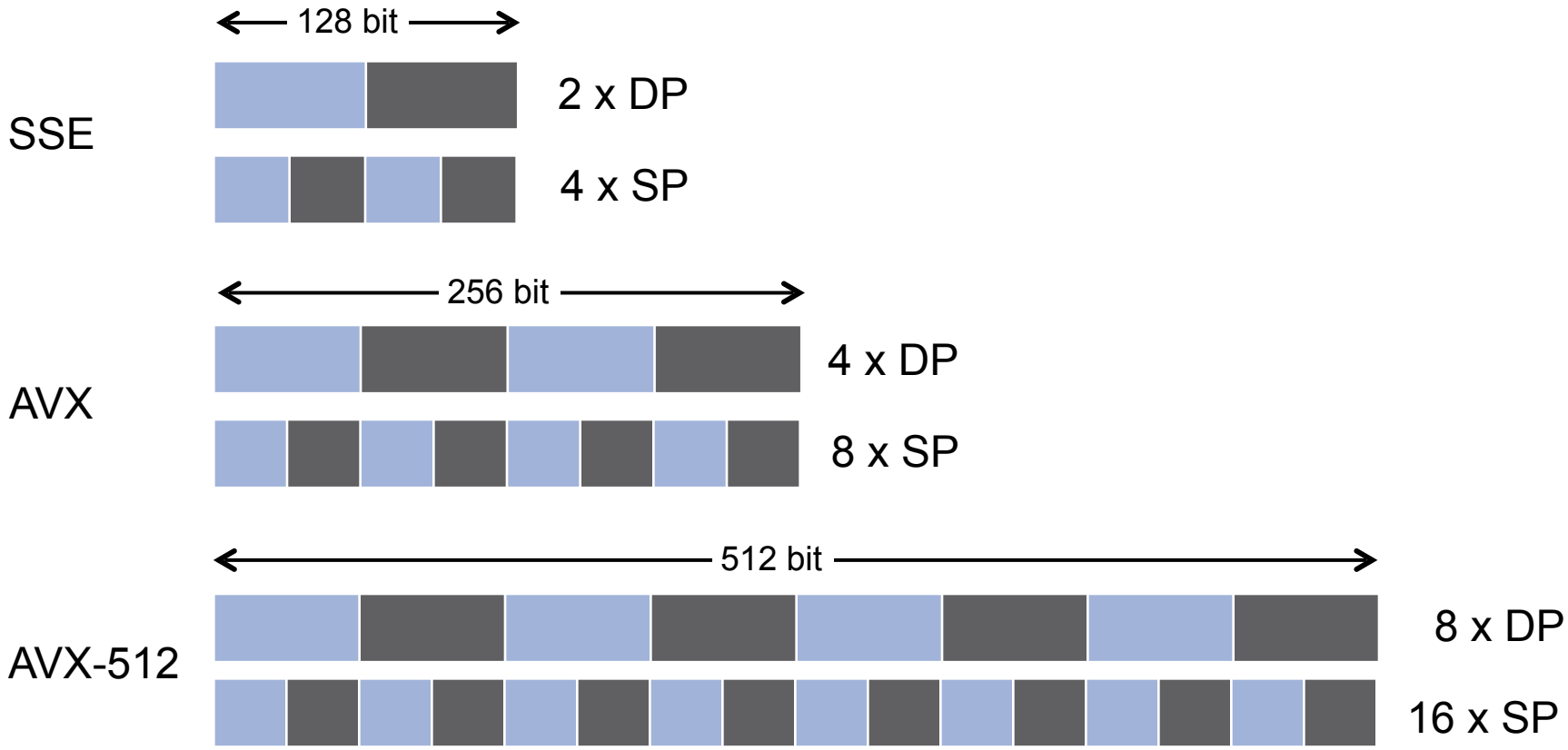
OpenMP

Topics

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

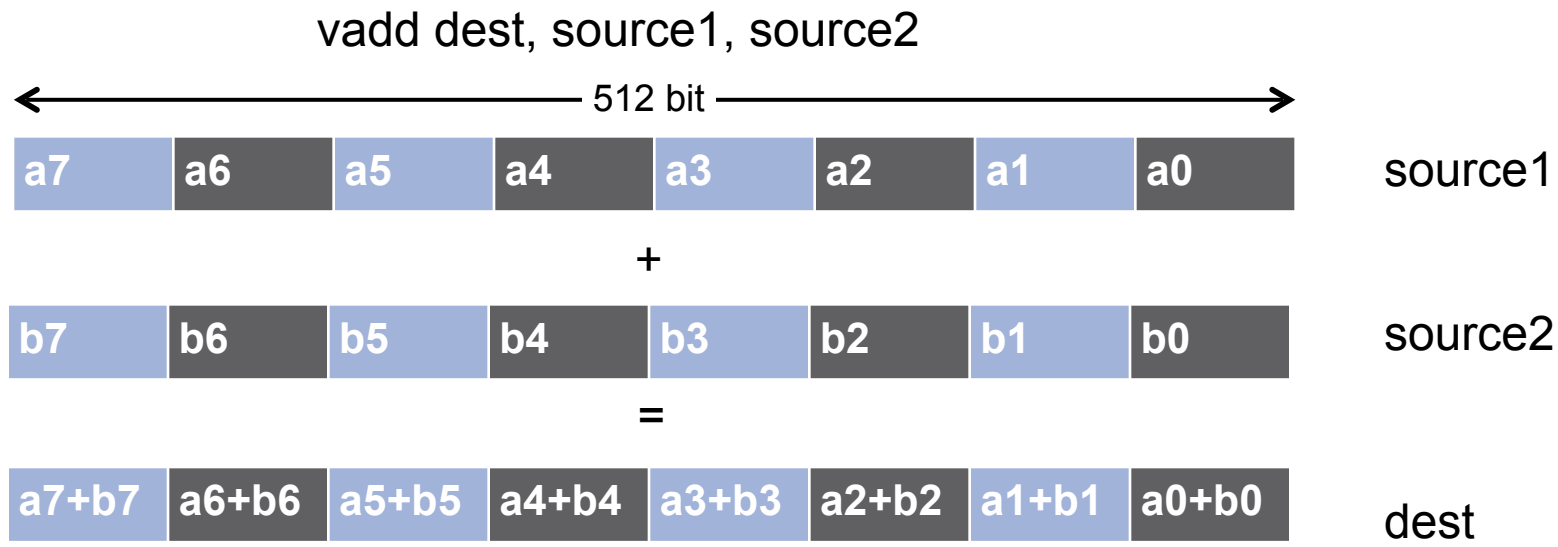
SIMD on x86 Architectures

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



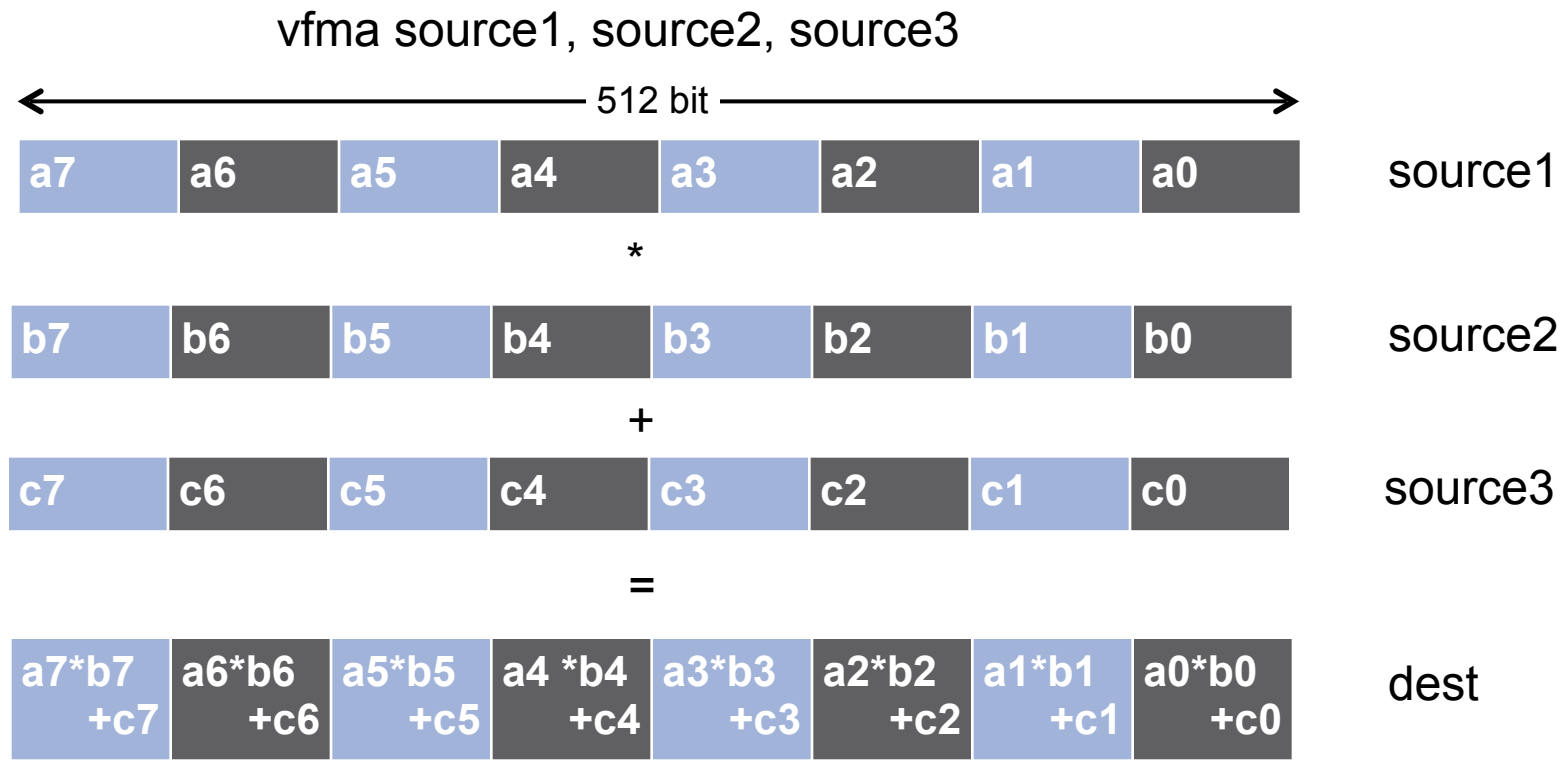
More Powerful SIMD Units

- SIMD instructions become more powerful



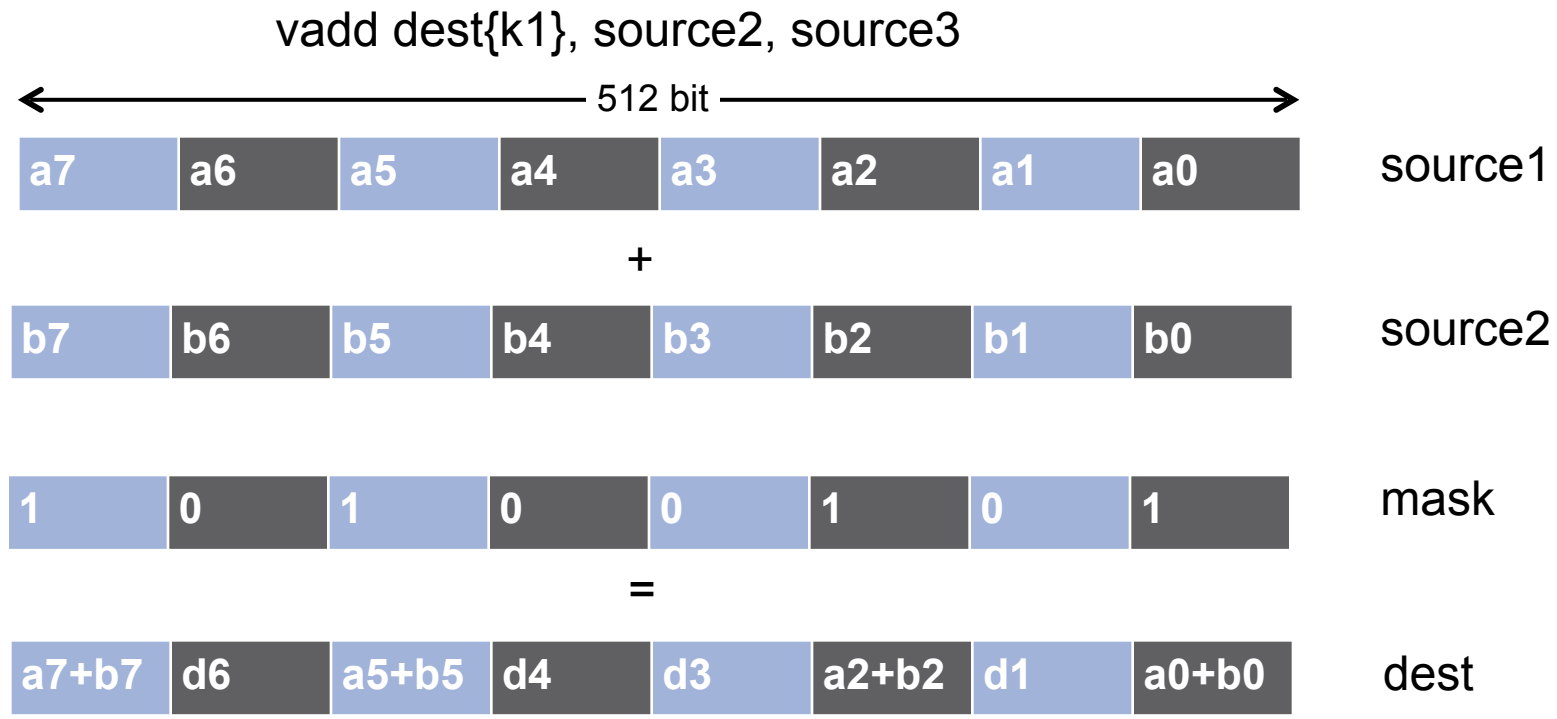
More Powerful SIMD Units

- SIMD instructions become more powerful



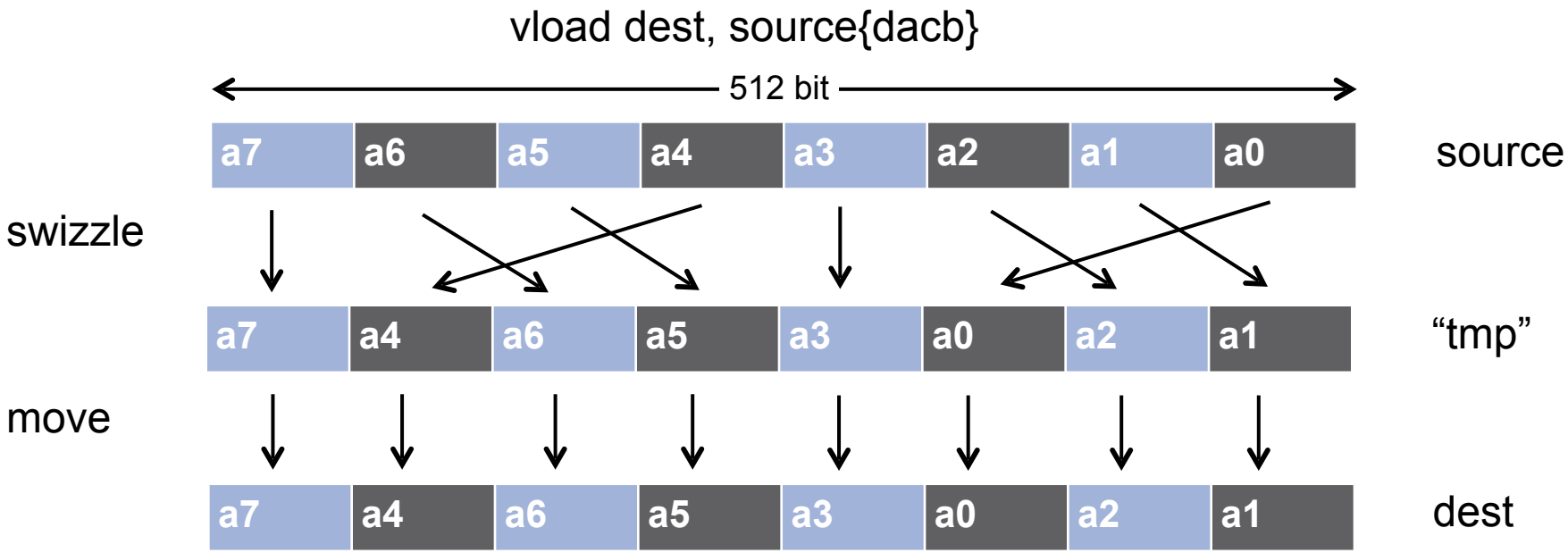
More Powerful SIMD Units

- 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];
    }
}
```

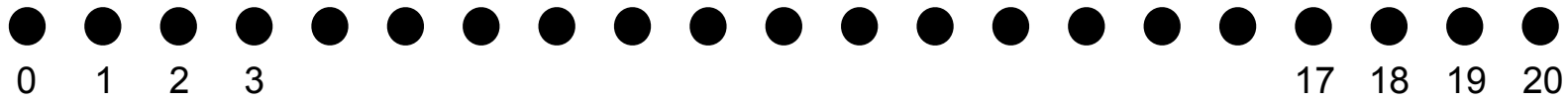
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- 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?

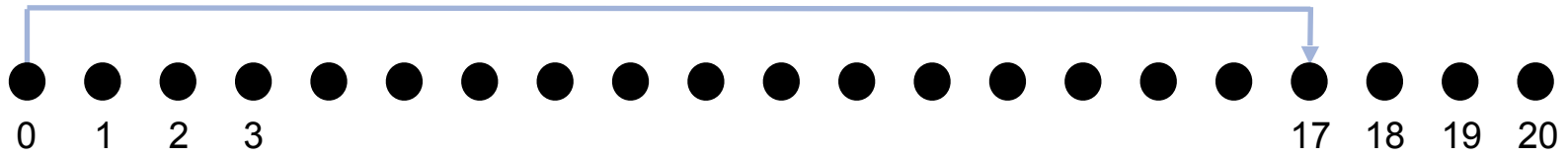
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

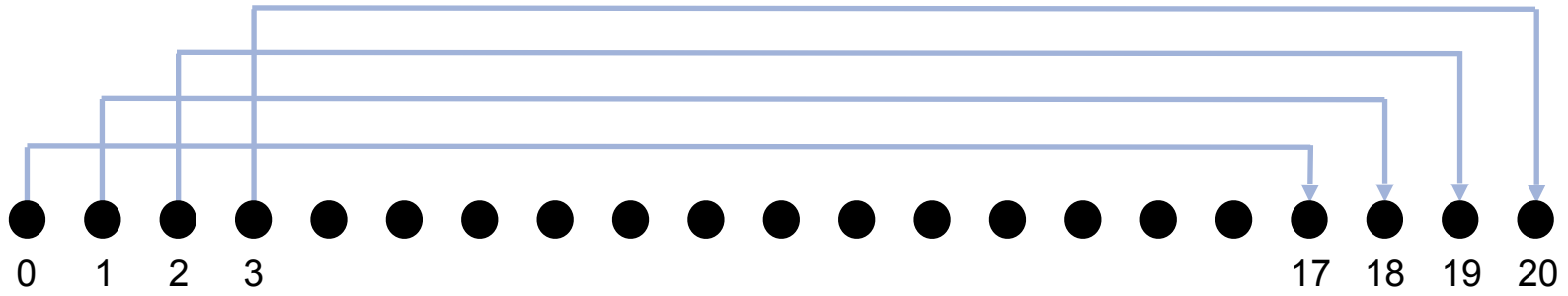
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

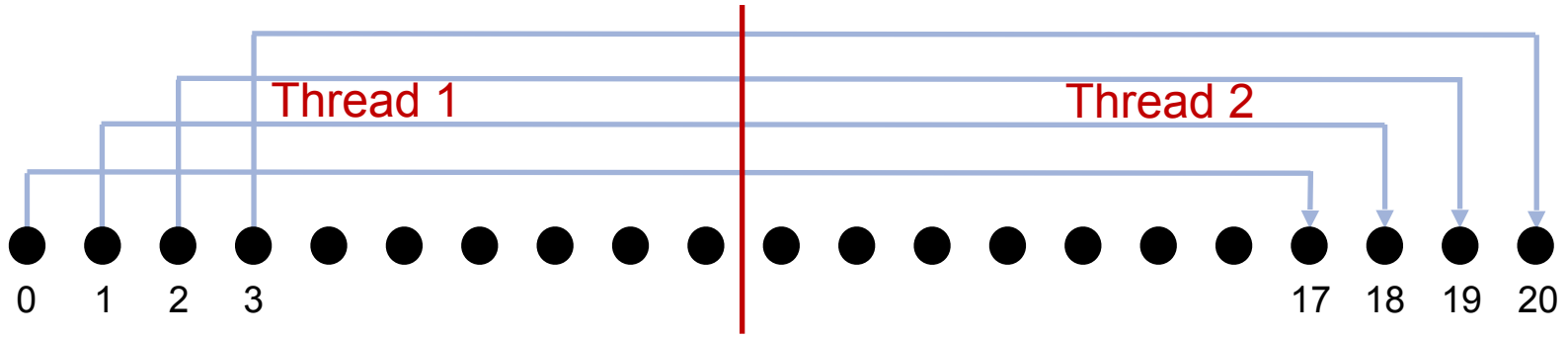
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

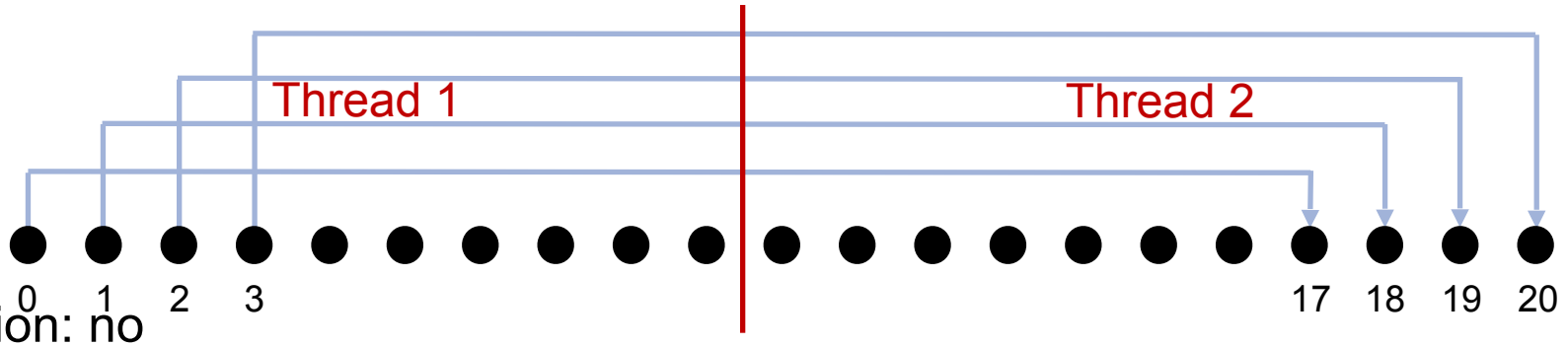
```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



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];
}   }
```



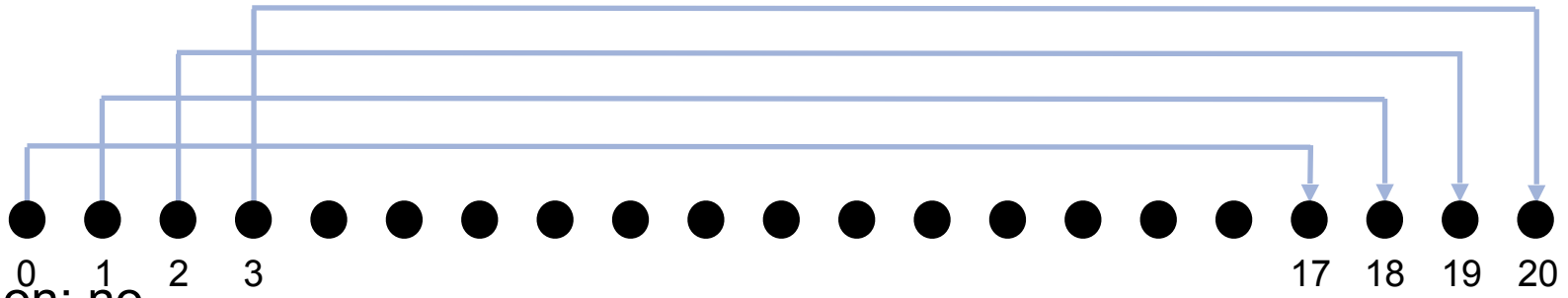
→ Parallelization: no

(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];
}   }
```



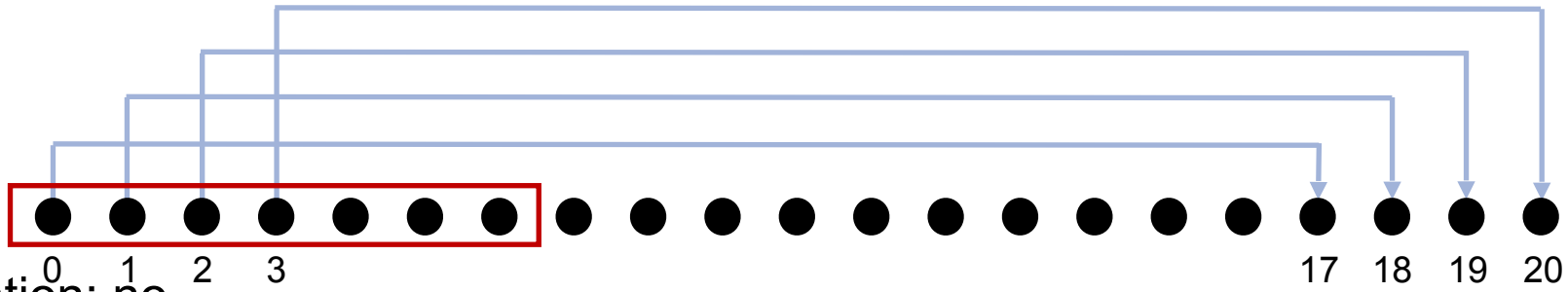
→ Parallelization: no

(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];
}   }
```



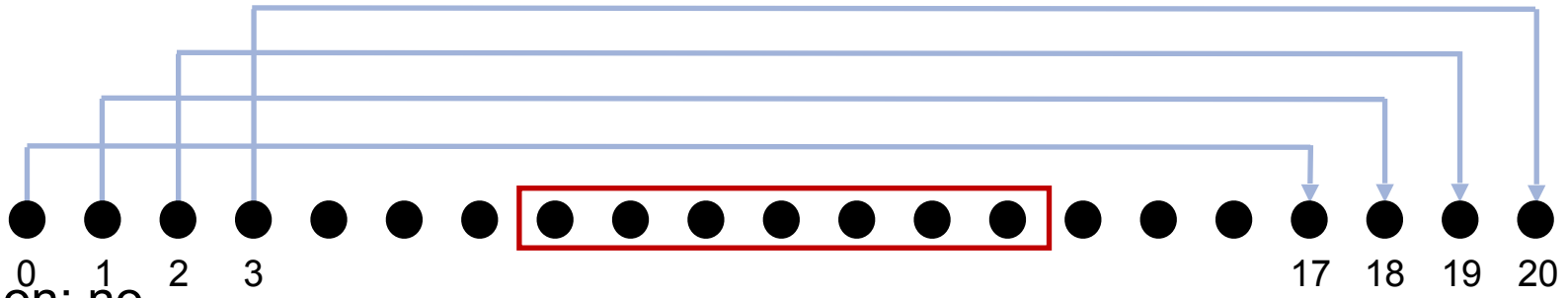
→ Parallelization: no

(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];
}   }
```



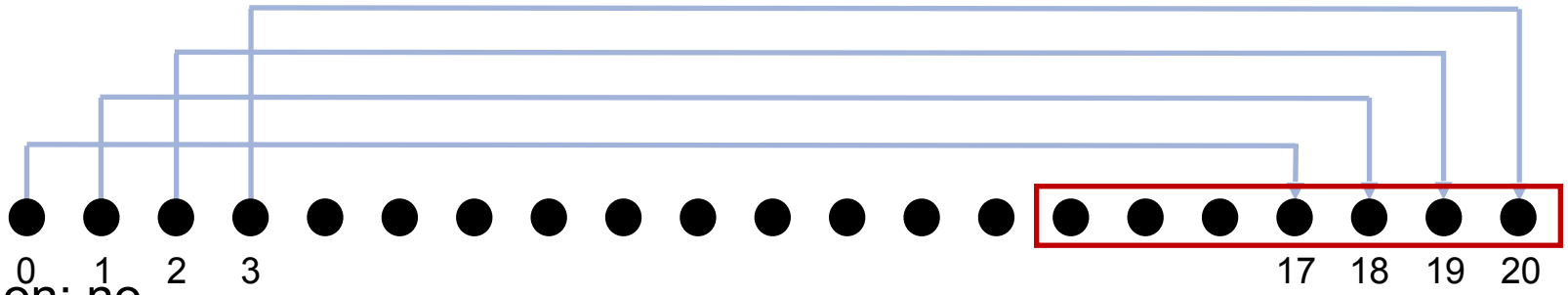
→ Parallelization: no

(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];
}   }
```



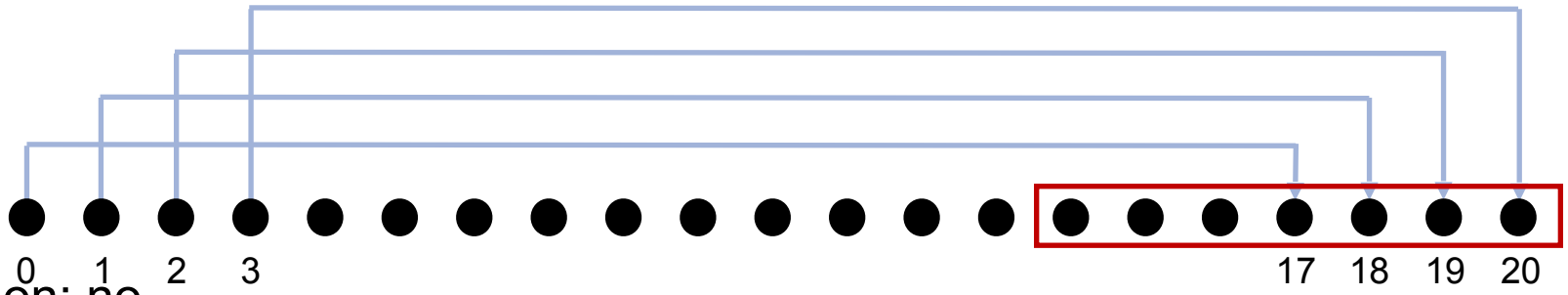
→ Parallelization: no

(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
```

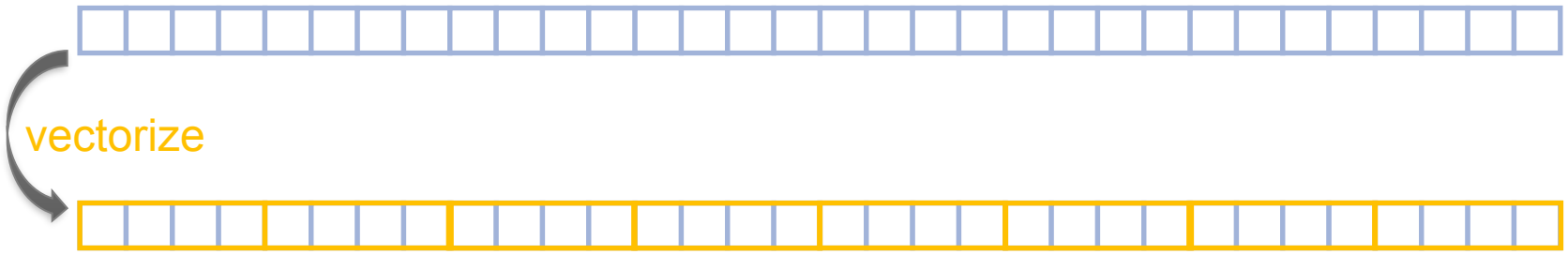
- Syntax (Fortran)

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

Example

Example

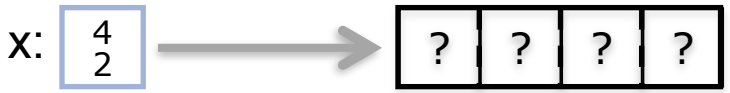
```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Data Sharing Clauses

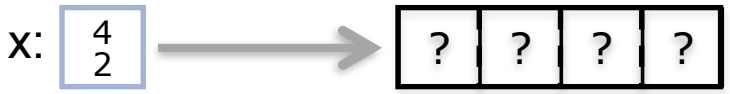
Data Sharing Clauses

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

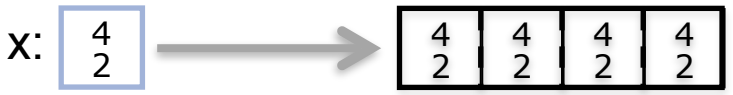


Data Sharing Clauses

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

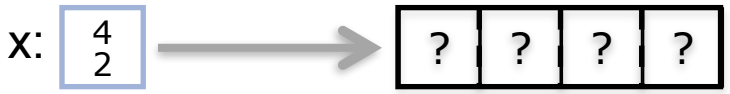


- `firstprivate(var-list):`
  Initialized vectors for variables in *var-list*

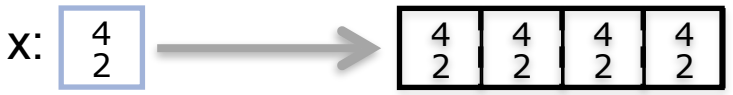


Data Sharing Clauses

- `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
```

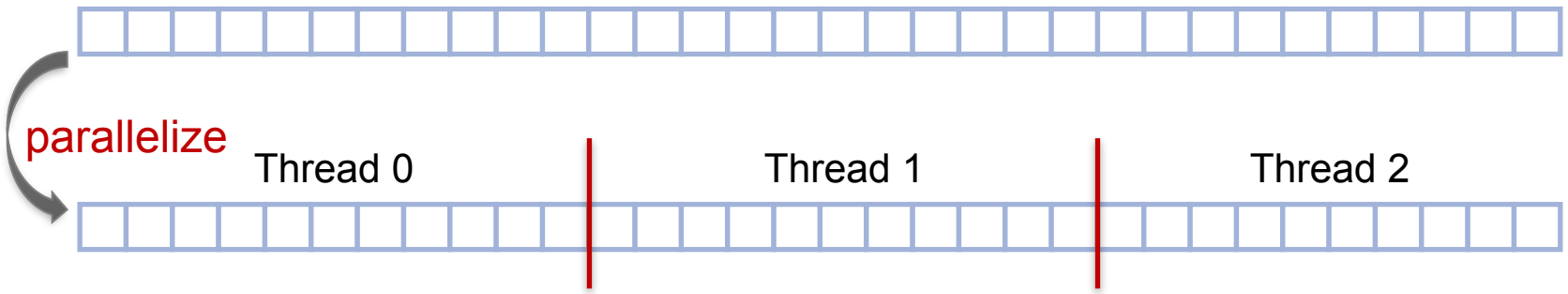
- Syntax (Fortran)

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

Example

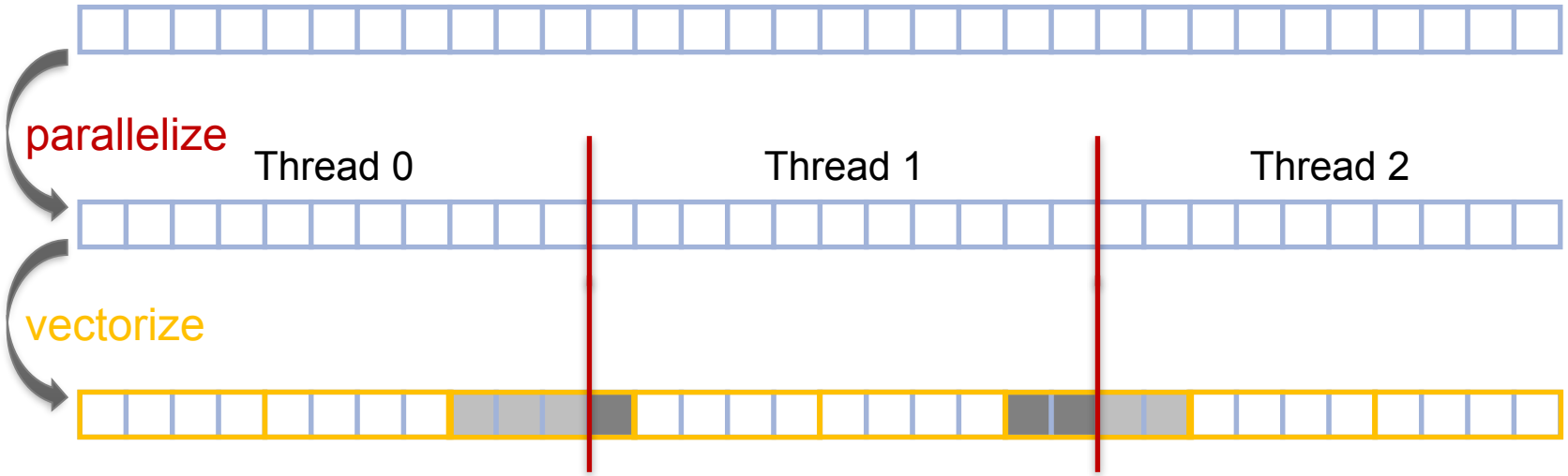
Example

```
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;
}
```



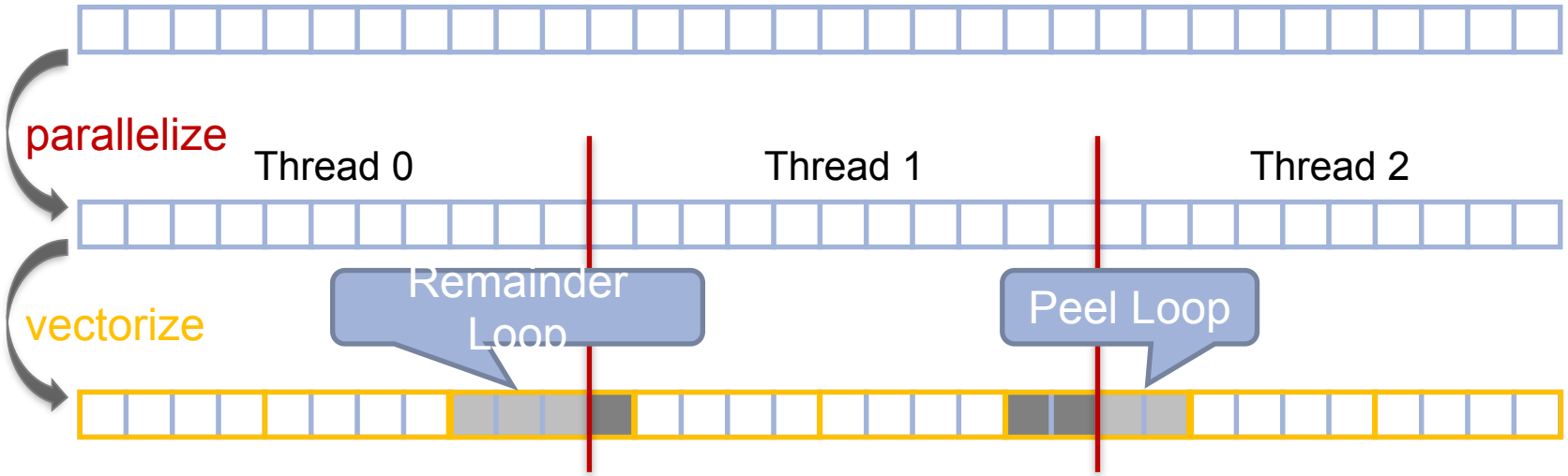
Example

```
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

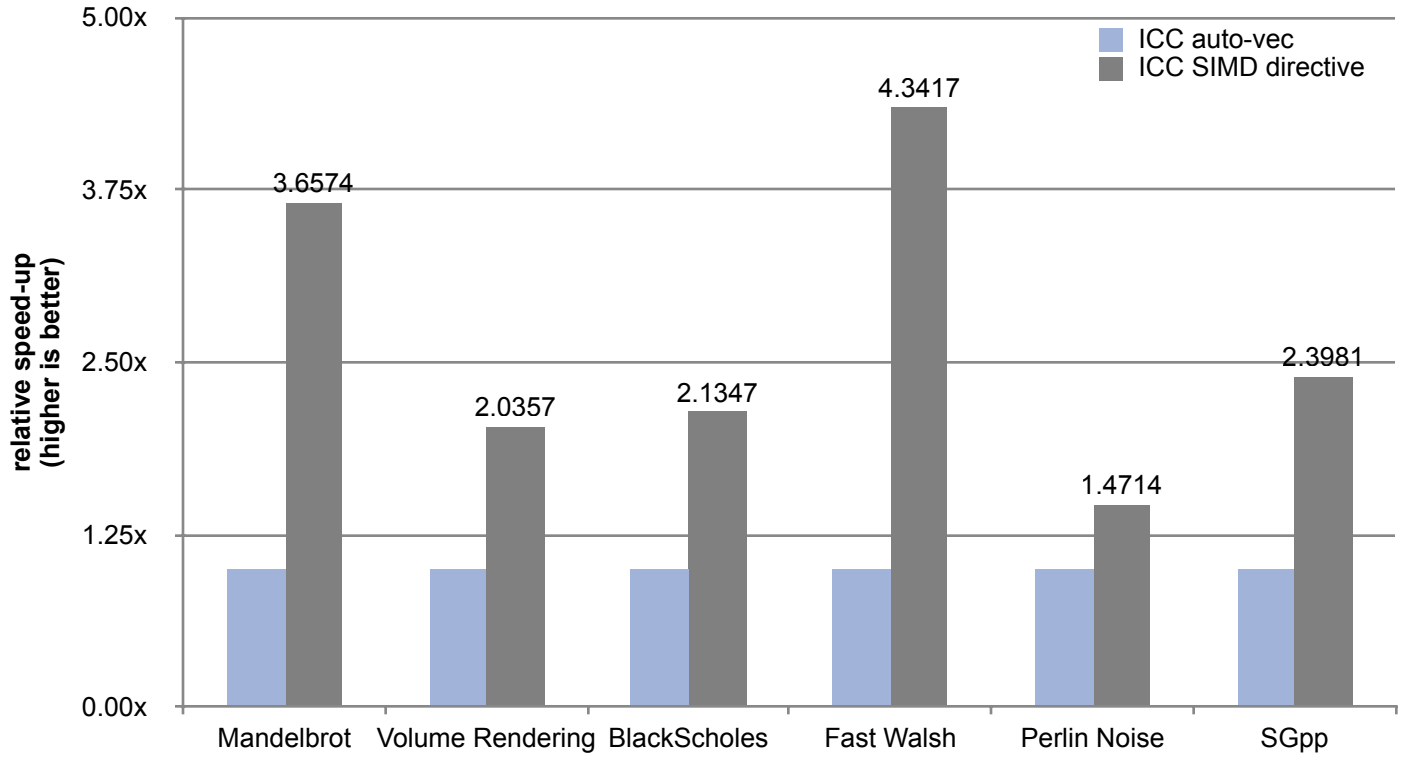
```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

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