Programming OpenMP

Christian Terboven

Michael Klemm

Agenda (in total 4 days)

- **Day 1: OpenMP Introduction**
  - Welcome
  - OpenMP Overview
  - Parallel Region
  - Worksharing
  - Scoping
  - Tasking (short introduction)
  - Vectorization / SIMD
  - Executing OpenMP programs
  - **Hands-On**

- Day 2: Tasking & Optimizations for NUMA
- Day 3: Introduction to Offloading with OpenMP
- Day 4: Advanced Offloading Topics

Material

https://github.com/cterboven/OpenMP-tutorial-CSC

Programming OpenMP

An Overview Of OpenMP

Christian Terboven

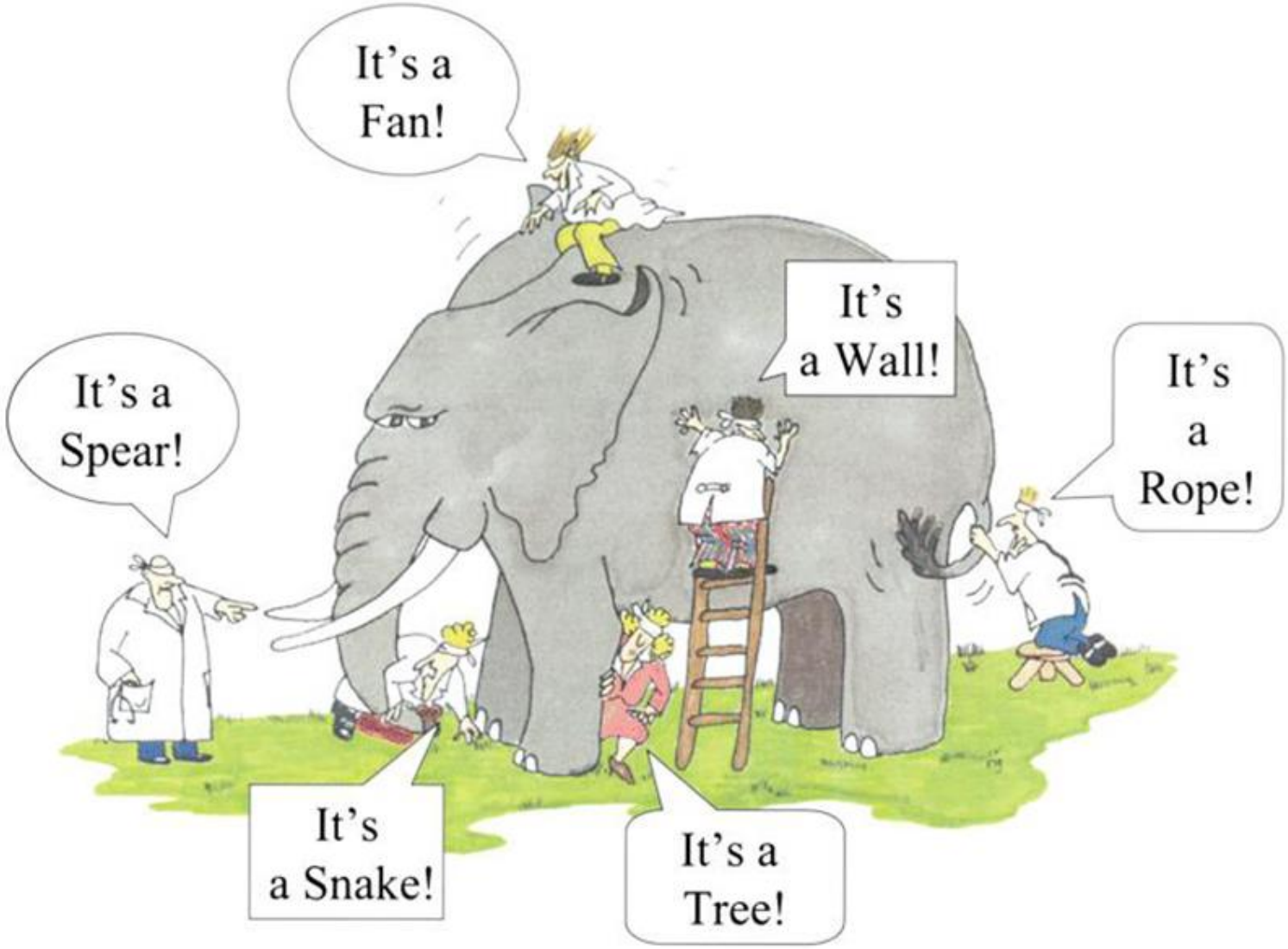
Michael Klemm

History

- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1
- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5
- 11/2018: OpenMP 5.0
- 11/2020: OpenMP 5.1

What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- …



Get your C/C++ and Fortran Reference Guide! Covers all of OpenMP 5.0!


OpenMP API 5.0 Page 1

OpenMP 5.0 API Syntax Reference Guide

openmp.org

The OpenMP® API is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran. OpenMP is suitable for a wide range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.

Functionality new/changed in OpenMP 5.0 is in this color, and in OpenMP 4.5 is in this color.
[n.n.n] Sections in the 5.0 spec. [n.n.n] Sections in the 4.5 spec. ● Deprecated in the 5.0 spec.

Directives and Constructs

An OpenMP executable directive applies to the succeeding structured block. A *structured-block* is an OpenMP construct or a block of executable statements with a single entry at the top and a single exit at the bottom. OpenMP directives except SIMD and **declare target** directives may not appear in **PURE** or **ELEMENTAL** procedures.

variant directives

Metadirectives [2.3.4]

A directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context.

```
C/C++
#pragma omp metadirective [clause[ [,] clause] ... ]
- or -
#pragma omp begin metadirective [clause[ [,] clause] ... ]
  stmt(s)
#pragma omp end metadirective
Fortran
!$omp metadirective [clause[ [,] clause] ... ]
- or -
!$omp begin metadirective [clause[ [,] clause] ... ]
  stmt(s)
!$omp end metadirective
```

clause:
when (*context-selector-specification: directive-variant*)
default (*directive-variant*)

declare variant [2.3.5]

Declares a specialized variant of a base function and the context in which it is used.

```
C/C++
#pragma omp declare variant(variant-func-id) clause
#pragma omp declare variant(variant-func-id) clause
[...]
  function definition or declaration
Fortran
!$omp declare variant ( &
  [base-proc-name :]variant-proc-name) clause
```

clause:
match (*context-selector-specification*)

variant-func-id: C/C++
The name of a function variant that is a base language identifier, or, for C++, a *template-id*.

variant-proc-name: for
The name of a function variant that is a base language identifier.

requires directive

requires [2.4]

Specifies the features an implementation must provide in order for the code to compile and to execute correctly.

```
C/C++
#pragma omp requires clause [ [,] clause] ... ]
Fortran
!$omp requires clause [ [,] clause] ... ]
```

clause:
reverse_offload
unified_address
unified_shared_memory
atomic_default_mem_order(seq_cst | acq_rel | relaxed)
dynamic_allocators

parallel construct

parallel [2.6] [2.5]

Creates a team of OpenMP threads that execute the region.

```
C/C++
#pragma omp parallel [clause[ [,] clause] ...]
  structured-block
Fortran
!$omp parallel [clause[ [,] clause] ...]
  structured-block
!$omp end parallel
```

clause:
private (*list*), firstprivate (*list*), shared (*list*)
copyin (*list*)
reduction ([*reduction-modifier,*] *reduction-identifier: list*)
proc_bind (master | close | spread)
allocate ([*allocator :*]*list*)
C/C++ if ([parallel :] *scalar-expression*)
C/C++ num_threads (*integer-expression*)
C/C++ default (shared | none)
For if ([parallel :] *scalar-logical-expression*)
For num_teams (*scalar-integer-expression*)
For default (shared | firstprivate | private | none)

teams construct

teams [2.7] [2.10.7]

Creates a league of thread teams where the master thread of each team executes the region.

```
C/C++
#pragma omp teams [clause[ [,] clause] ...]
  structured-block
Fortran
!$omp teams [clause[ [,] clause] ...]
  structured-block
!$omp end teams
```

clause:
private (*list*), firstprivate (*list*), shared (*list*)
reduction ([default ,] *reduction-identifier : list*)
allocate ([*allocator :*]*list*)
C/C++ num_teams (*integer-expression*)
C/C++ thread_limit (*integer-expression*)
C/C++ default (shared | none)
For num_teams (*scalar-integer-expression*)
For thread_limit (*scalar-integer-expression*)
For default (shared | firstprivate | private | none)

Worksharing constructs

sections [2.8.1] [2.7.2]

A noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

```
C/C++
#pragma omp sections [clause[ [,] clause] ...]
{
[#pragma omp section]
  structured-block
[#pragma omp section
  structured-block]
...
}
Fortran
!$omp sections [clause[[,] clause] ...]
[!$omp section]
  structured-block
[!$omp section
  structured-block]
...
!$omp end sections [nowait]
```

clause:
private (*list*), firstprivate (*list*)
lastprivate ([*lastprivate-modifier:*] *list*)
reduction ([*reduction-modifier,*] *reduction-identifier: list*)
allocate ([*allocator :*]*list*)
C/C++ nowait

single [2.8.2] [2.7.3]

Specifies that the associated structured block is executed by only one of the threads in the team.

```
C/C++
#pragma omp single [clause[ [,] clause] ...]
  structured-block
Fortran
!$omp single [clause[ [,] clause] ...]
  structured-block
!$omp end single [end_clause[ [,] end_clause] ...]
```

clause:
private (*list*), firstprivate (*list*)
allocate ([*allocator :*]*list*)
C/C++ copyprivate (*list*)
C/C++ nowait

end_clause: for
copyprivate (*list*), nowait

workshare [2.8.3] [2.7.4]

Divides the execution of the enclosed structured block into separate units of work, each executed only once by one thread.

```
Fortran
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

Worksharing-loop construct

for / do [2.9.2] [2.7.1]

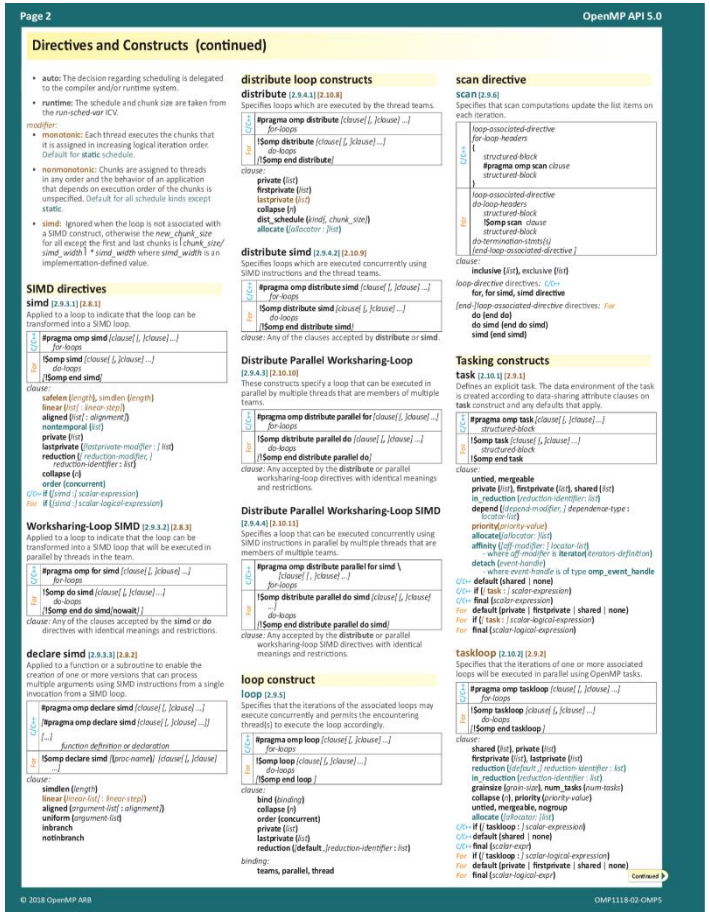
Specifies that the iterations of associated loops will be executed in parallel by threads in the team.

```
C/C++
#pragma omp for [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp do [clause[ [,] clause] ...]
  do-loops
!$omp end do [nowait] ]
```

clause:
private (*list*), firstprivate (*list*)
lastprivate ([*lastprivate-modifier:*] *list*)
linear ([*list* : *linear-step*])
schedule ([*modifier* [, *modifier*] :] *kind*[, *chunk_size*])
collapse (*n*), ordered ([*n*])
allocate ([*allocator :*]*list*)
order (concurrent)
C/C++ reduction ([*reduction-modifier,*] *reduction-identifier: list*)
C/C++ nowait

kind:
- **static:** Iterations are divided into chunks of size *chunk_size* and assigned to threads in the team in round-robin fashion in order of thread number.
- **dynamic:** Each thread executes a chunk of iterations then requests another chunk until none remain.
- **guided:** Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. Chunk size is different for each chunk, with each successive chunk smaller than the last.

Continued


Page 2 OpenMP API 5.0

Directives and Constructs (continued)

- **auto:** The decision regarding scheduling is delegated to the compiler and/or runtime system.
- **runtime:** The schedule and chunk size are taken from the *run-sched-var* ICV.

modifier:
- **monotonic:** Each thread executes the chunks that it is assigned in increasing logical iteration order. Default for **static** schedule.
- **nonmonotonic:** Chunks are assigned to threads in any order and the behavior of an application that depends on execution order of the chunks is unspecified. Default for all schedule kinds except **static**.
- **simd:** Ignored when the loop is not associated with a SIMD construct, otherwise the *new_chunk_size* for all except the first and last chunks is (*chunk_size*/ *simd_width*) * *simd_width* where *simd_width* is an implementation-defined value.

SIMD directives

simd [2.9.3.1] [2.8.1]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

```
C/C++
#pragma omp simd [clause[ [,] clause]...]
  for-loops
Fortran
!$omp simd [clause[ [,] clause] ...]
  do-loops
[!$omp end simd]
```

clause:
safelen (*length*), simdlen (*length*)
linear ([*list* [: *linear-step*]])
aligned ([*list* [: *alignment*]])
nontemporal (*list*)
private ([*list*])
lastprivate ([*lastprivate-modifier* :] *list*)
reduction ([*reduction-modifier,*] *reduction-identifier* : *list*)
collapse (*n*)
order (concurrent)
C/C++ if ([*simd :*] *scalar-expression*)
For if ([*simd :*] *scalar-logical-expression*)

Worksharing-Loop SIMD [2.9.3.2] [2.8.3]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop that will be executed in parallel by threads in the team.

```
C/C++
#pragma omp for simd [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp do simd [clause[ [,] clause] ...]
  do-loops
[!$omp end do simd[nowait] ]
```

clause: Any of the clauses accepted by the **simd** or **do** directives with identical meanings and restrictions.

declare simd [2.9.3.3] [2.8.2]

Applied to a function or a subroutine to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.

```
C/C++
#pragma omp declare simd [clause[ [,] clause] ...]
#pragma omp declare simd [clause[ [,] clause] ...]
[...]
  function definition or declaration
Fortran
!$omp declare simd ([proc-name]) [clause[ [,] clause]
...]
```

clause:
simdlen (*length*)
linear ([*linear-list* [: *linear-step*]])
aligned ([*argument-list* [: *alignment*]])
uniform (*argument-list*)
inbranch
notinbranch

distribute loop constructs

distribute [2.9.4.1] [2.10.8]

Specifies loops which are executed by the thread teams.

```
C/C++
#pragma omp distribute [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp distribute [clause[ [,] clause] ...]
  do-loops
[!$omp end distribute]
```

clause:
private (*list*)
firstprivate (*list*)
lastprivate (*list*)
collapse (*n*)
dist_schedule (*kind*[, *chunk_size*])
allocate ([*allocator :*]*list*)

distribute simd [2.9.4.2] [2.10.9]

Specifies loops which are executed concurrently using SIMD instructions and the thread teams.

```
C/C++
#pragma omp distribute simd [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp distribute simd [clause[ [,] clause] ...]
  do-loops
[!$omp end distribute simd]
```

clause: Any of the clauses accepted by **distribute** or **simd**.

Distribute Parallel Worksharing-Loop [2.9.4.3] [2.10.10]

These constructs specify a loop that can be executed in parallel by multiple threads that are members of multiple teams.

```
C/C++
#pragma omp distribute parallel for [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp distribute parallel do [clause[ [,] clause] ...]
  do-loops
[!$omp end distribute parallel do]
```

clause: Any accepted by the **distribute** or parallel worksharing-loop directives with identical meanings and restrictions.

Distribute Parallel Worksharing-Loop SIMD [2.9.4.4] [2.10.11]

Specifies a loop that can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams.

```
C/C++
#pragma omp distribute parallel for simd \
  [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp distribute parallel do simd [clause[ [,] clause]
...]
  do-loops
[!$omp end distribute parallel do simd]
```

clause: Any accepted by the **distribute** or parallel worksharing-loop SIMD directives with identical meanings and restrictions.

loop construct

loop [2.9.5]

Specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.

```
C/C++
#pragma omp loop [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp loop [clause[ [,] clause] ...]
  do-loops
[!$omp end loop ]
```

clause:
bind (*binding*)
collapse (*n*)
order (concurrent)
private (*list*)
lastprivate (*list*)
reduction ([default ,]*reduction-identifier* : *list*)

binding:
teams, parallel, thread

scan directive

scan [2.9.6]

Specifies that scan computations update the list items on each iteration.

```
C/C++
loop-associated-directive
for-loop-headers
{
  structured-block
  #pragma omp scan clause
  structured-block
}
Fortran
loop-associated-directive
do-loop-headers
  structured-block
  !$omp scan clause
  structured-block
do-termination-stmt(s)
[end-loop-associated-directive ]
```

clause:
inclusive (*list*), exclusive (*list*)

loop-directive directives: C/C++
for, for simd, simd directive

[end-]loop-associated-directive directives: For
do (end do)
do simd (end do simd)
simd (end simd)

Tasking constructs

task [2.10.1] [2.9.1]

Defines an explicit task. The data environment of the task is created according to data-sharing attribute clauses on task construct and any defaults that apply.

```
C/C++
#pragma omp task [clause[ [,] clause] ...]
  structured-block
Fortran
!$omp task [clause[ [,] clause] ...]
  structured-block
!$omp end task
```

clause:
untied, mergeable
private (*list*), firstprivate (*list*), shared (*list*)
in_reduction (*reduction-identifier: list*)
depend ([*depend-modifier,*] *dependence-type* : *locator-list*)
priority(*priority-value*)
allocate([*allocator:*]*list*)
affinity ([*aff-modifier:*] *locator-list*)
- where *aff-modifier* is **iterator**(*iterators-definition*)

detach (*event-handle*)
- where *event-handle* is of type **omp_event_handle**

C/C++ default (shared | none)
C/C++ if ([task :] *scalar-expression*)
C/C++ final (*scalar-expression*)
For default (private | firstprivate | shared | none)
For if ([task :] *scalar-logical-expression*)
For final (*scalar-logical-expression*)

taskloop [2.10.2] [2.9.2]

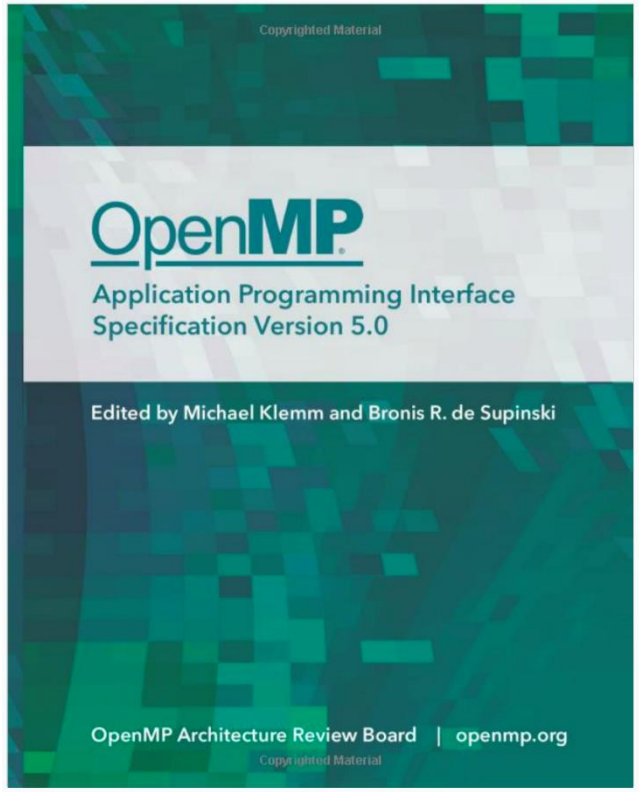
Specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks.

```
C/C++
#pragma omp taskloop [clause[ [,] clause] ...]
  for-loops
Fortran
!$omp taskloop [clause[ [,] clause] ...]
  do-loops
[!$omp end taskloop ]
```

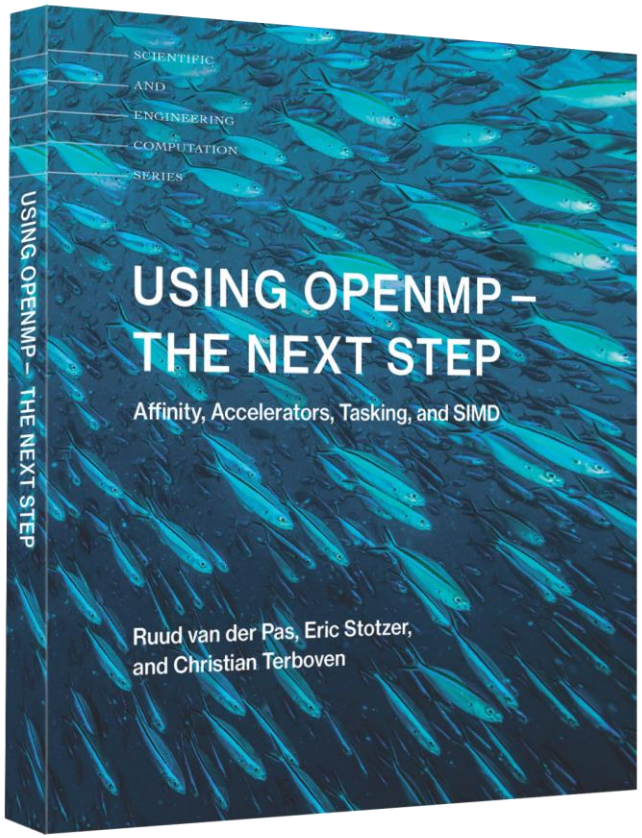
clause:
shared (*list*), private (*list*)
firstprivate (*list*), lastprivate (*list*)
reduction ([default ,] *reduction-identifier* : *list*)
in_reduction (*reduction-identifier : list*)
grainsize (*grain-size*), num_tasks (*num-tasks*)
collapse (*n*), priority (*priority-value*)
untied, mergeable, nogroup
allocate ([*allocator :*]*list*)
C/C++ if ([taskloop :] *scalar-expression*)
C/C++ default (shared | none)
C/C++ final (*scalar-expr*)
For if ([taskloop :] *scalar-logical-expression*)
For default (private | firstprivate | shared | none)
For final (*scalar-logical-expr*)

Continued

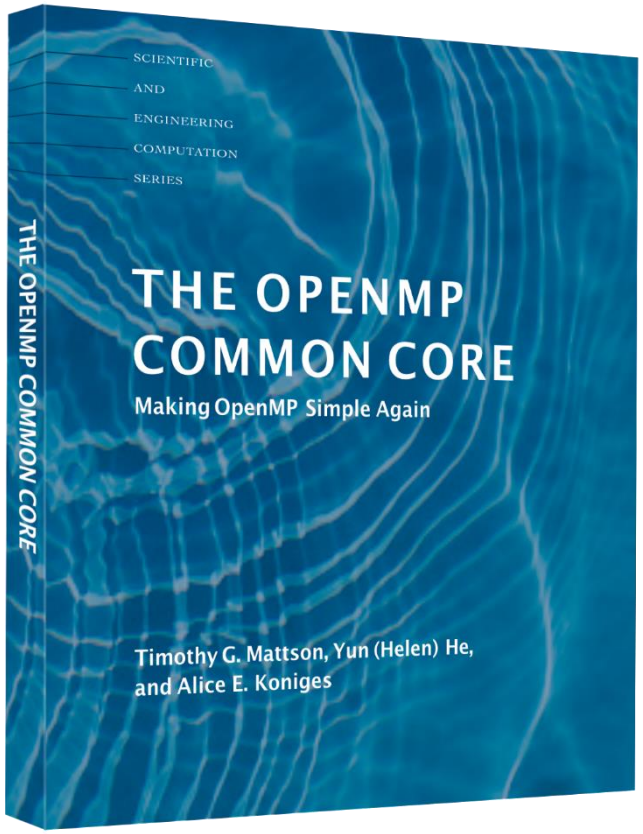
Recent Books About OpenMP



A printed copy of the 5.0 specifications, 2019



A book that covers all of the OpenMP 4.5 features, 2017



A new book about the OpenMP Common Core, 2019

Programming OpenMP

Parallel Region

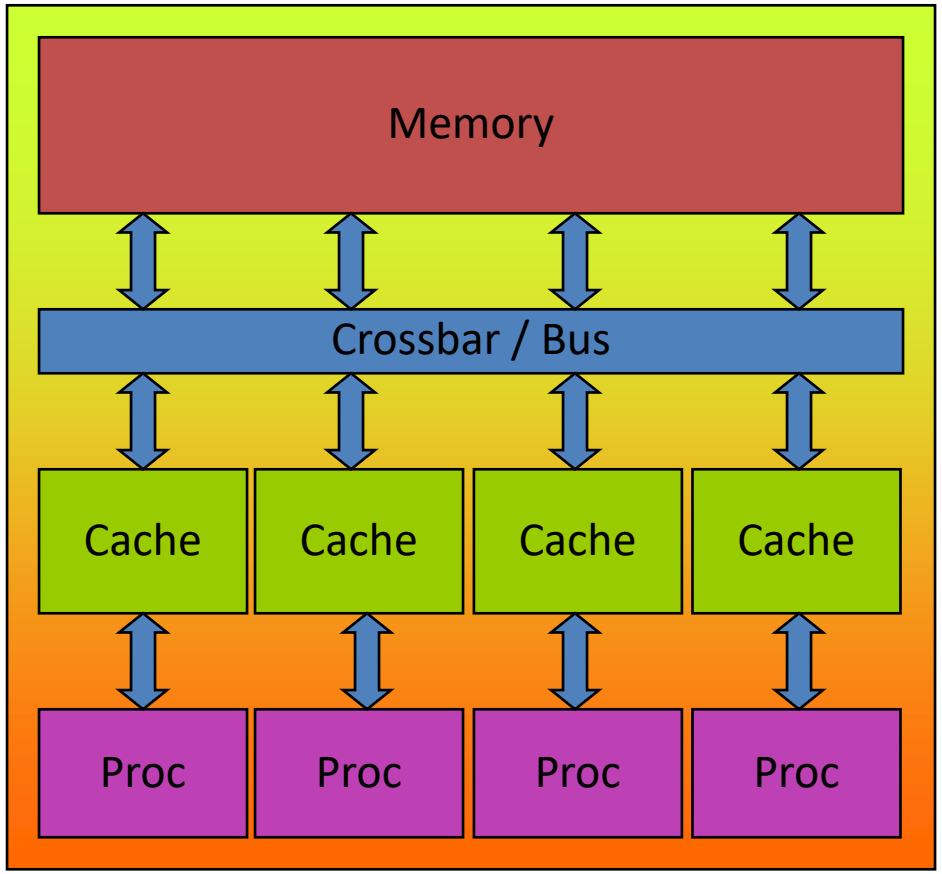
Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

OpenMP‘s machine model

- OpenMP: Shared-Memory Parallel Programming Model.



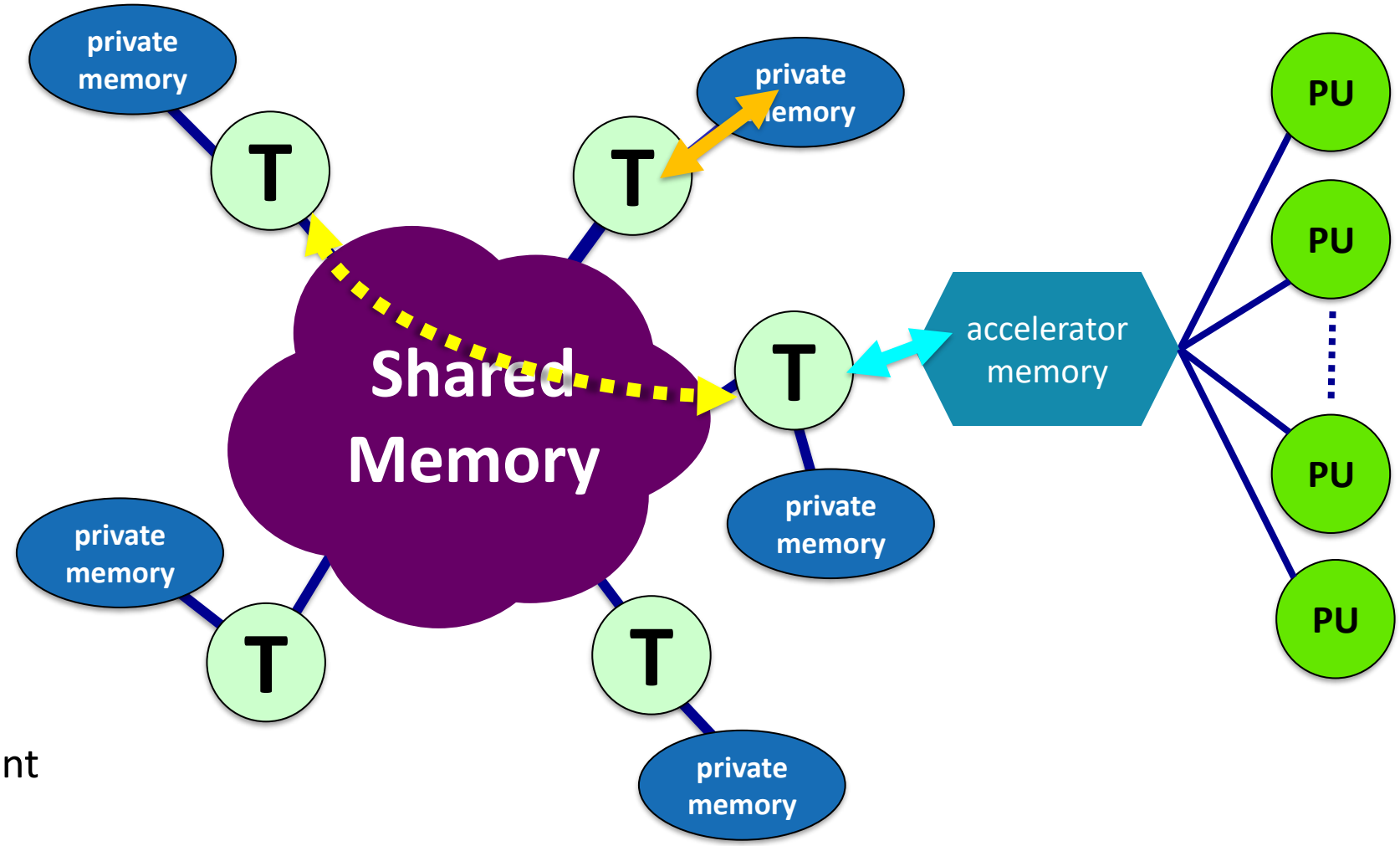
All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.

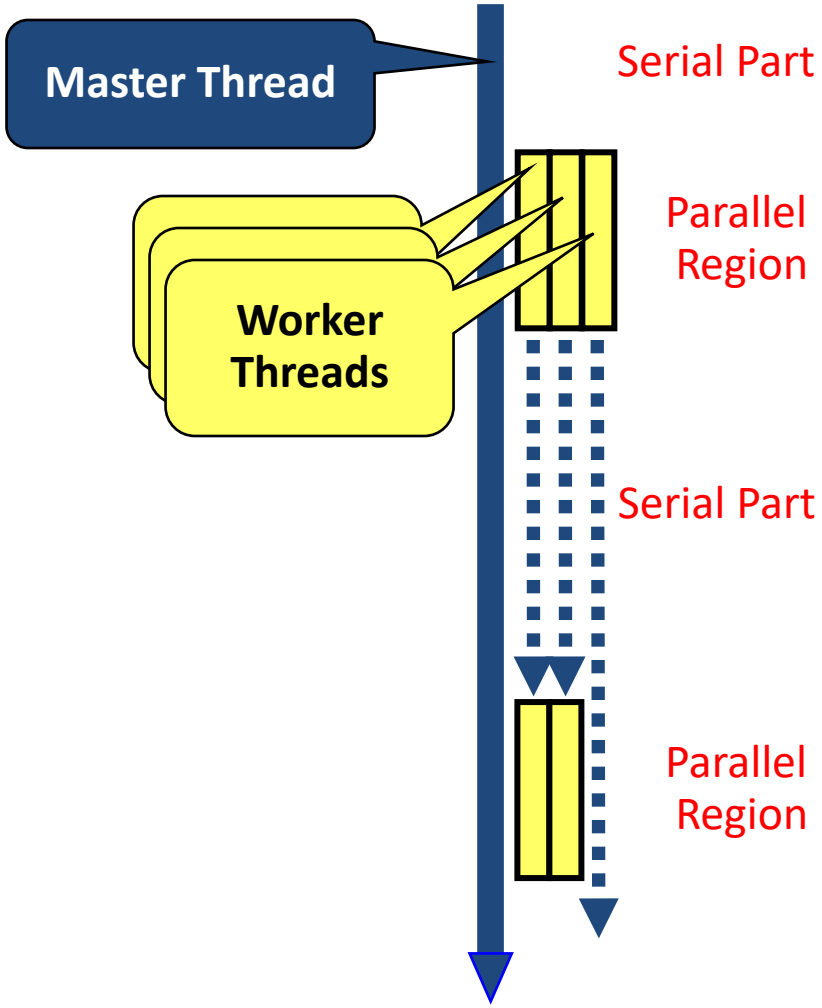
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application



The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

Fortran

```
!$omp parallel
    ...
    structured block
    ...
!$omp end parallel
```

- *Structured Block*
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- *Specification of number of threads:*
  - Environment variable: `OMP_NUM_THREADS=…`
  - Or: Via `num_threads` clause: add `num_threads(num)` to the parallel construct

Starting OpenMP Programs on Linux

- From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4
./program
```

- From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4    ./program
```

Hello OpenMP World

Programming OpenMP

Worksharing

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

For Worksharing

- If only the *parallel* construct is used, each thread executes the Structured Block.
- Program Speedup: *Worksharing*
- OpenMP's most common Worksharing construct: *for*

C/C++

```
int i;
#pragma omp for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

Fortran

```
INTEGER :: i
!$omp do
DO i = 0, 99
   a[i] = b[i] + c[i]
END DO
```

  - Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.

- Loops often account for most of a program's runtime!

Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

Thread 2

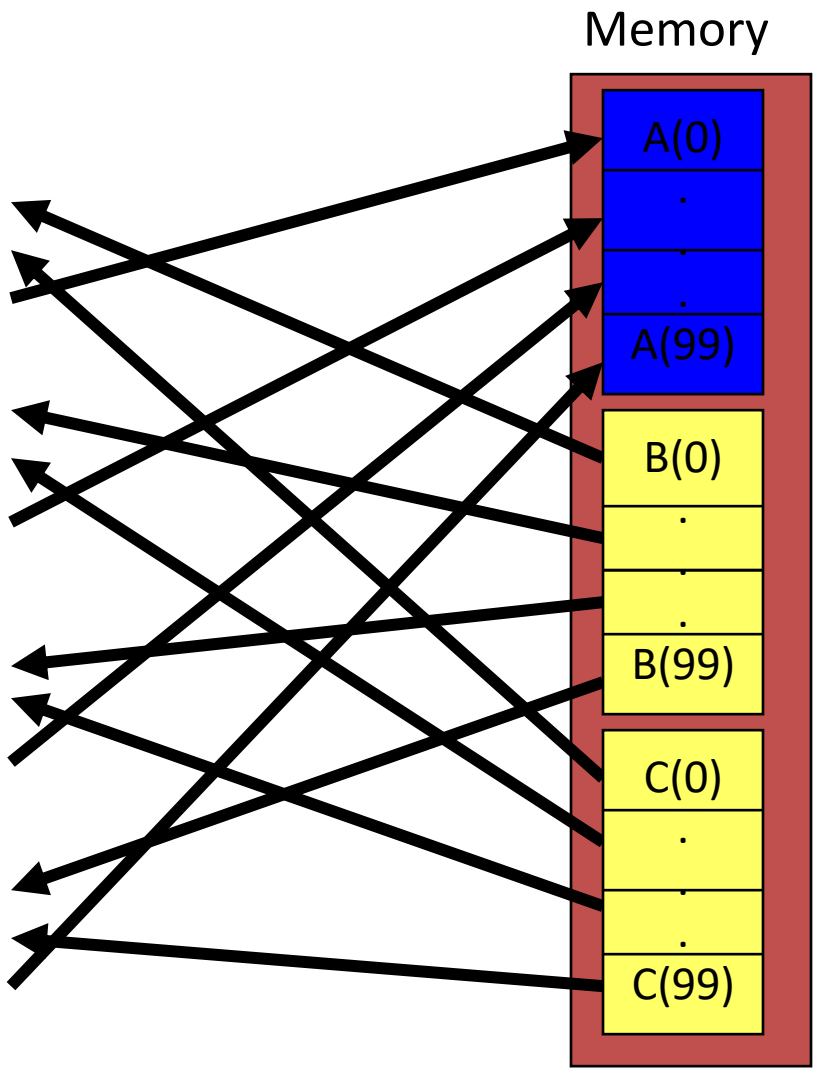
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```



The Barrier Construct

- OpenMP `barrier` (implicit or explicit)
  - Threads wait until all threads of the current *Team* have reached the barrier

C/C++

```
#pragma omp barrier
```

- All worksharing constructs contain an implicit barrier at the end

The Single Construct

C/C++

```
#pragma omp single [clause]
... structured block ...
```

Fortran

```
!$omp single [clause]
... structured block ...
!$omp end single
```

- The `single` construct specifies that the enclosed structured block is executed by only on thread of the team.
  - It is up to the runtime which thread that is.

- Useful for:
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see later…

The Master Construct

C/C++

```
#pragma omp master[clause]
... structured block ...
```

Fortran

```
!$omp master[clause]
... structured block ...
!$omp end master
```

- The `master` construct specifies that the enclosed structured block is executed only by the master thread of a team.

- Note: The master construct is no worksharing construct and does not contain an implicit barrier at the end.

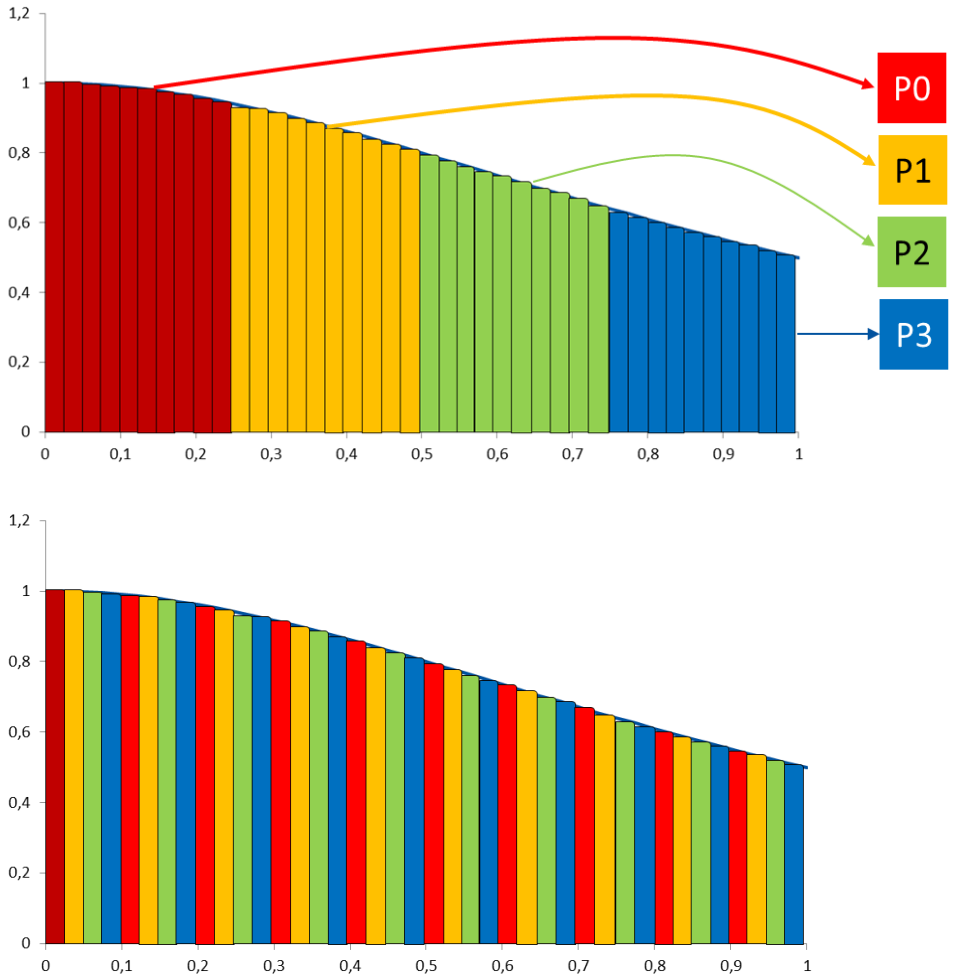
Vector Addition

Influencing the For Loop Scheduling / 1

- *for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- Default is `schedule(static)`.

Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - → `schedule(static [, chunk])`
  - → Decomposition depending on chunksize
  - → Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**
  - → No/low runtime overhead
- **Cons?**
  - → No dynamic workload balancing



Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
- Pros ?
  - Workload distribution
- Cons?
  - Runtime Overhead
  - Chunk size essential for performance
  - No NUMA optimizations possible

Synchronization Overview

- Can all loops be parallelized with `for`-constructs? No!
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent.
    BUT: This test alone is not sufficient:

C/C++

```
int i, int s = 0;

#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

- *Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).

Synchronization: Critical Region

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
   ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
#pragma omp critical
   {  s = s + a[i];  }
}
```

Programming OpenMP

Scoping

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

Tasks are introduced later

Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

Really: try to avoid the use of threadprivate and static variables!

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

Back to our example

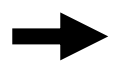
C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
#pragma omp critical
   {  s = s + a[i];  }
}
```

It‘s your turn: Make It Scale!

```
#pragma omp parallel
{

#pragma omp for
   for (i = 0; i < 99; i++)
   {

         s   = s    + a[i];

   }

} // end parallel
```

```
do i = 0, 99
    s = s + a(i)
end do
```



```
do i = 0, 24
    s = s + a(i)
end do
```

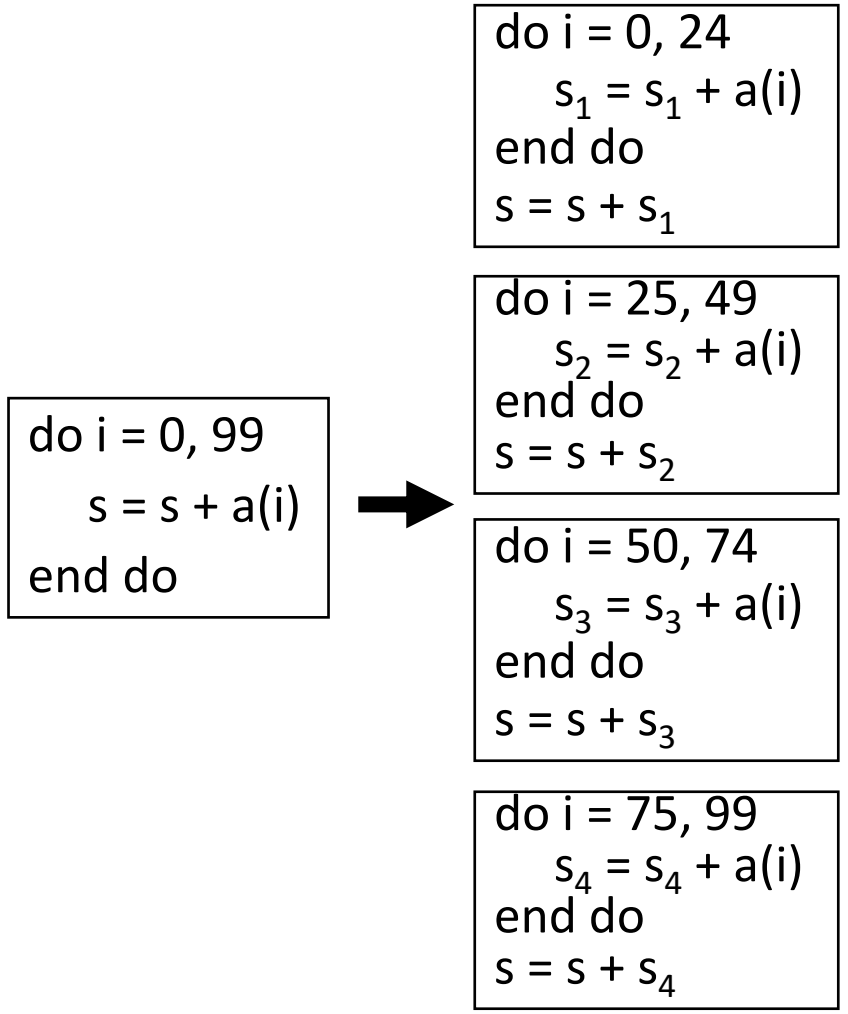
```
do i = 25, 49
    s = s + a(i)
end do
```

```
do i = 50, 74
    s = s + a(i)
end do
```

```
do i = 75, 99
    s = s + a(i)
end do
```

(done)

```
#pragma omp parallel
{
    double ps = 0.0;    // private variable
#pragma omp for
   for (i = 0; i < 99; i++)
   {
         ps = ps + a[i];
   }
#pragma omp critical
{
    s += ps;
}
} // end parallel
```



The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
  - `reduction(operator:list)`
  - The result is provided in the associated reduction variable

C/C++

```
int i, s = 0;

#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
   s = s + a[i];
}
```

  - Possible reduction operators with initialization value:
    `+ (0), * (1), - (0), & (~0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)`
  - Remark: OpenMP also supports user-defined reductions (not covered here)

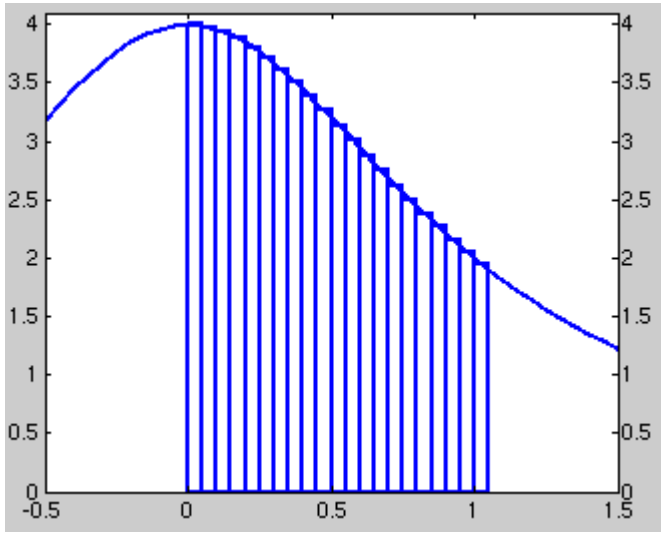
Example

PI

Example: Pi (1/2)

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

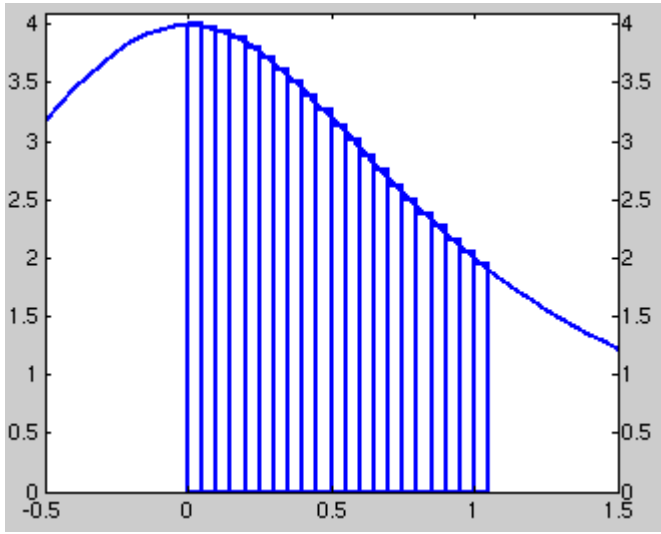
$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Example: Pi (2/2)

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

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



Programming OpenMP

OpenMP Tasking Introduction

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

What is a Task in OpenMP?

- Tasks are work units whose execution
  - → may be deferred or…
  - → … can be executed immediately
- Tasks are composed of
  - → **code** to execute, a **data** environment (initialized at creation time), internal **control** variables (ICVs)
- Tasks are created…
  - … when reaching a parallel region → implicit tasks are created (per thread)
  - … when encountering a task construct → explicit task is created
  - … when encountering a taskloop construct → explicit tasks per chunk are created
  - … when encountering a target construct → target task is created

Tasking Execution Model

- Supports unstructured parallelism
  - → unbounded loops

```
while ( <expr> ) {
   ...
}
```

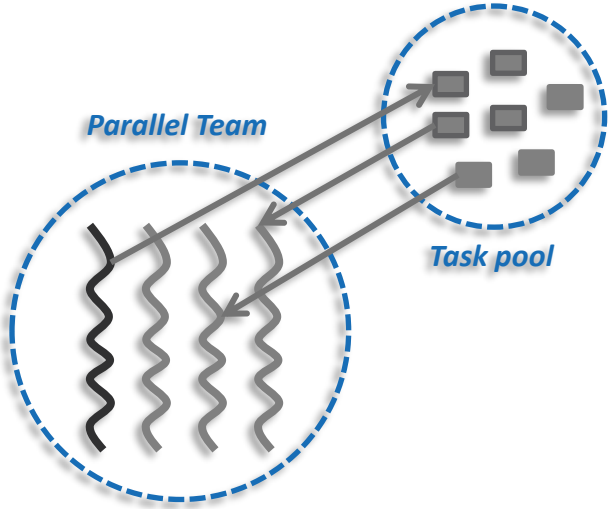
  - → recursive functions

```
void myfunc( <args> )
{
   ...; myfunc( <newargs> ); ...;
}
```

- Several scenarios are possible:
  - → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```



OpenMP Tasking Idiom

- OpenMP programmers need a specific idiom to kick off task-parallel execution: `parallel master`
  - → OpenMP version 5.0 introduced the `parallel master` construct
  - → With OpenMP version 5.1 this becomes `parallel masked`

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

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

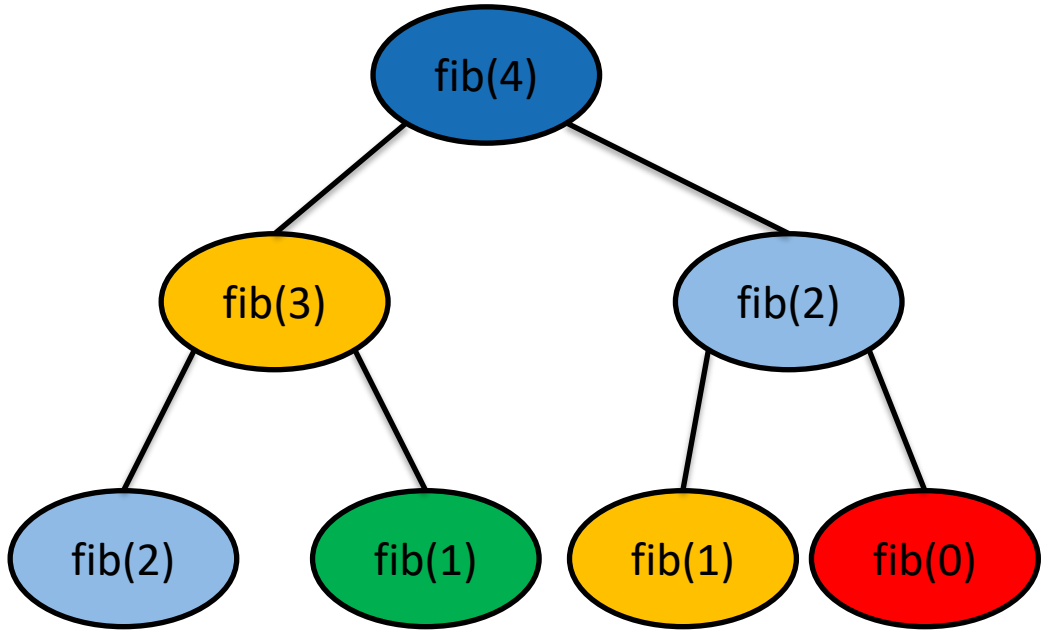
Fibonacci Numbers (in a Stupid Way ☺)

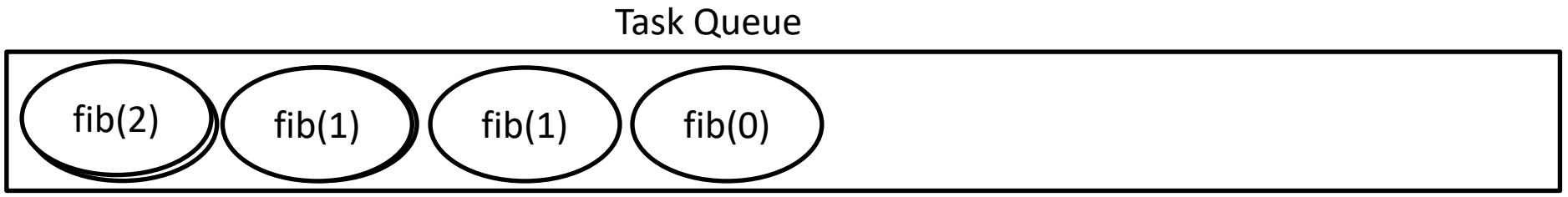
```
 1  int main(int argc,
 2           char* argv[])
 3  {
 4      [...]
 5      #pragma omp parallel
 6      {
 7          #pragma omp master
 8          {
 9              fib(input);
10          }
11      }
12      [...]
13  }
```

```
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  }
```

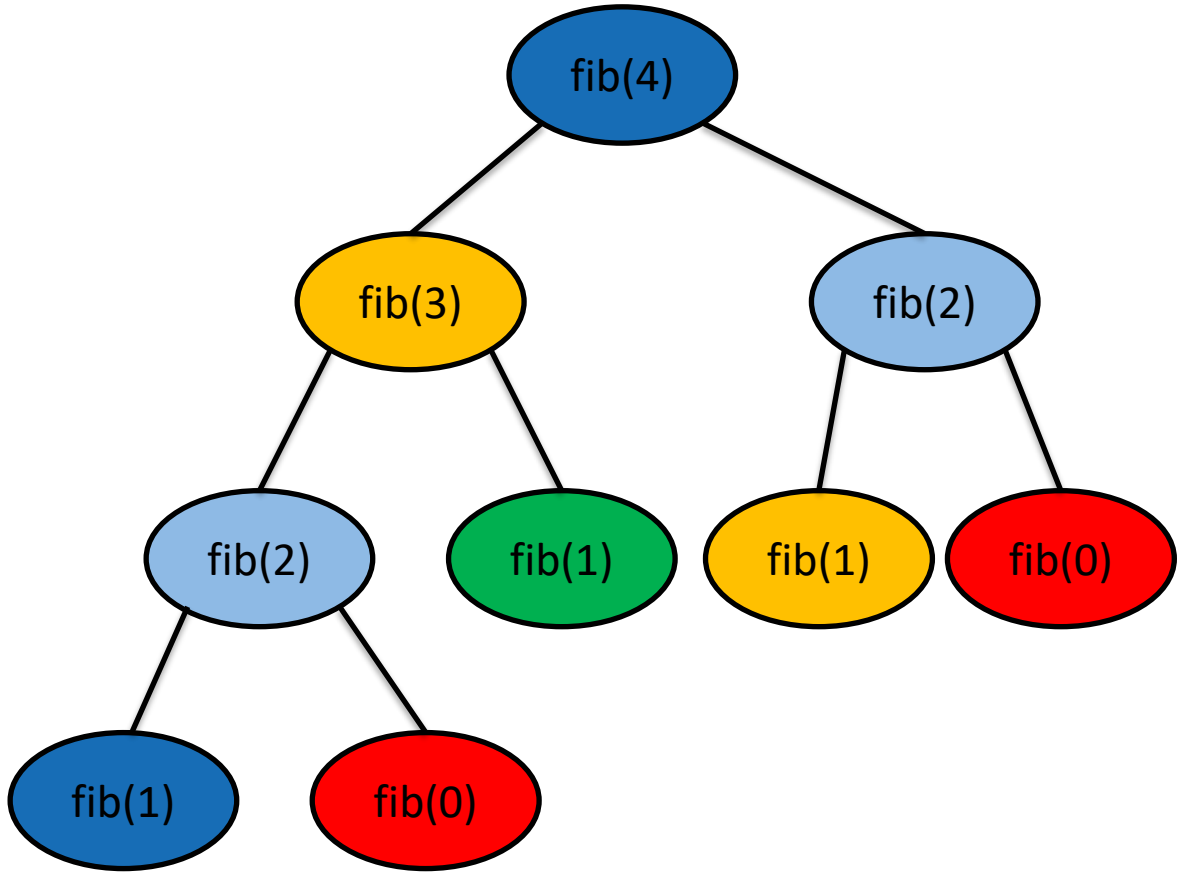
- Only one thread enters `fib()` from `main()`.
- That thread creates the two initial work tasks and starts the parallel recursion.
- The `taskwait` construct is required to wait for the result for `x` and `y` before the task can sum up.

- 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





- 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
- …



Advanced OpenMP Tutorial

Performance: Vectorization

Christian Terboven

Michael Klemm

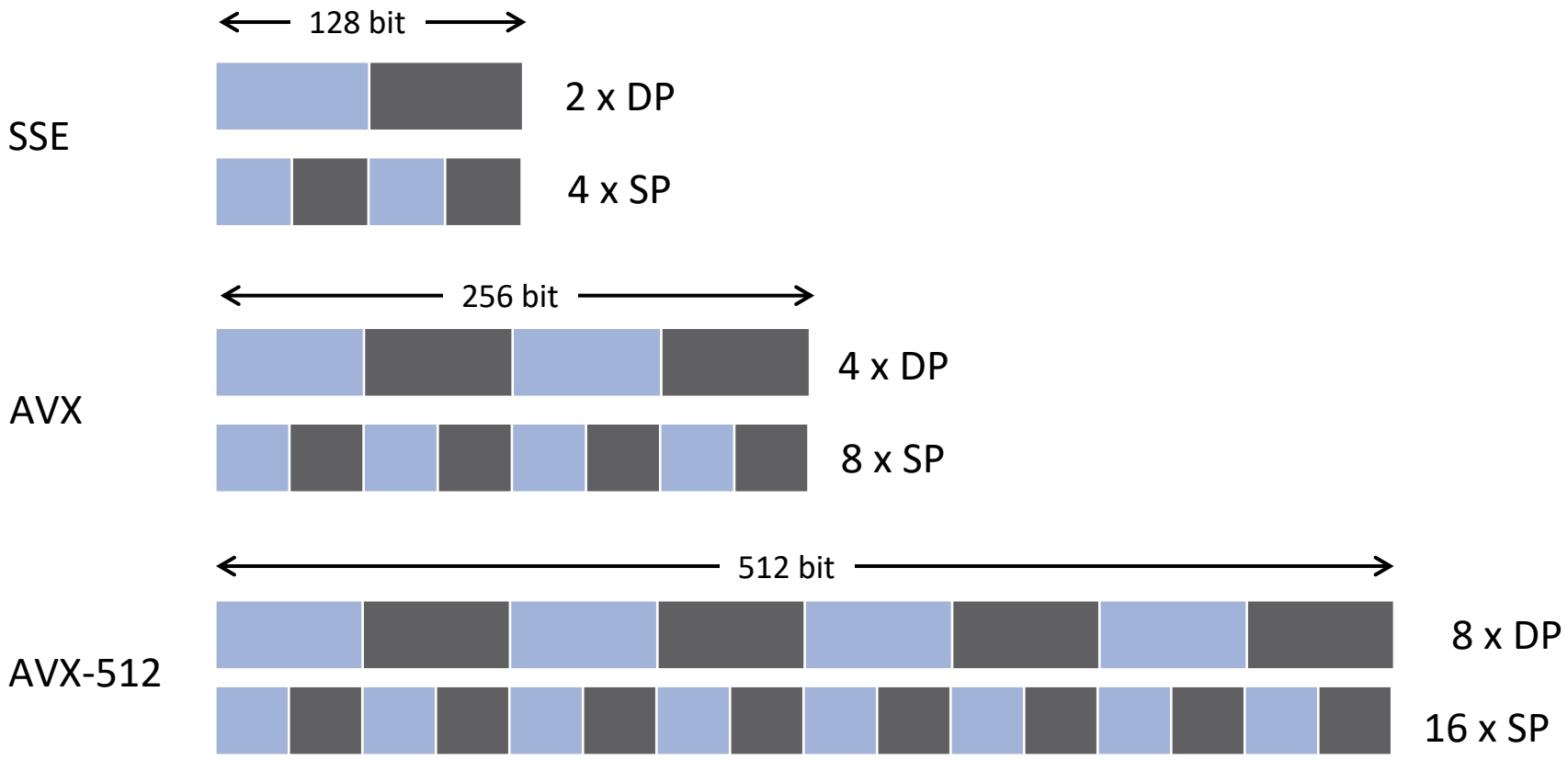
Ruud van der Pas

Topics

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

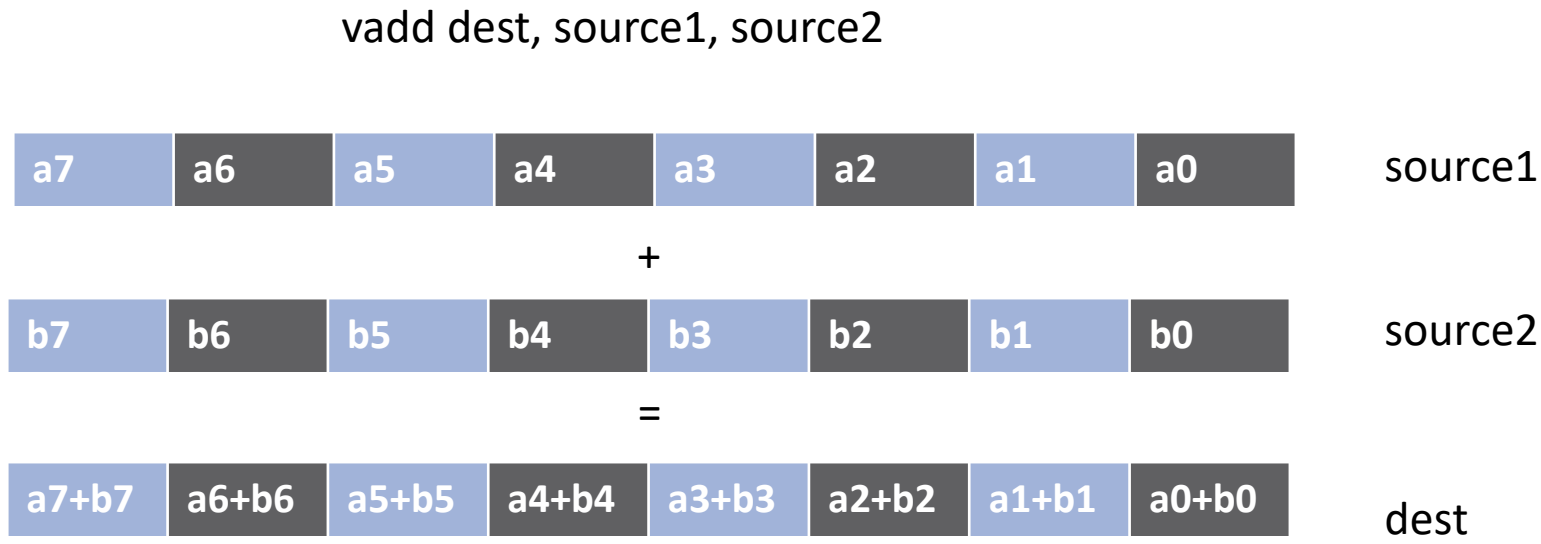
SIMD on x86 Architectures

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



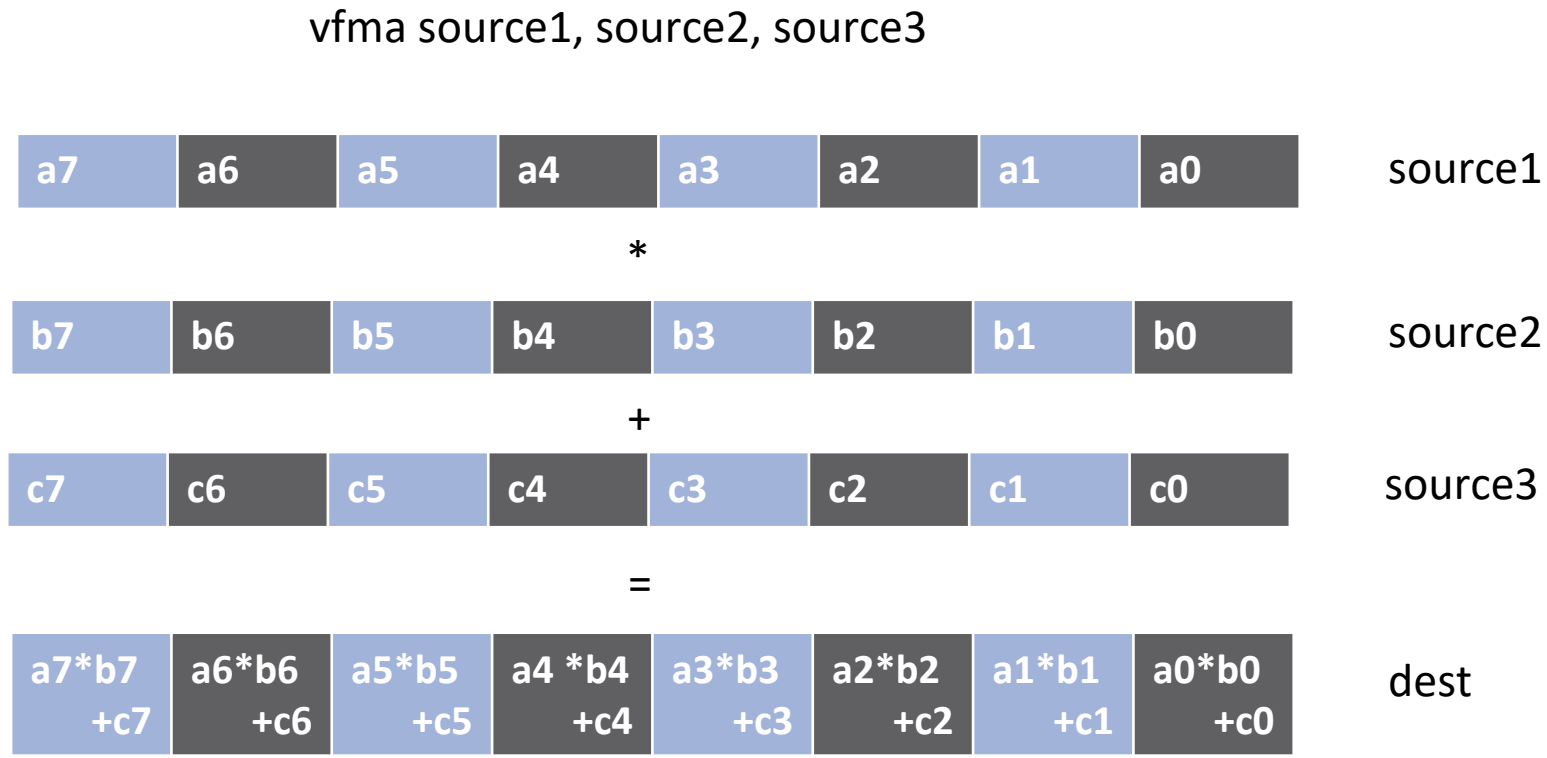
More Powerful SIMD Units

- SIMD instructions become more powerful



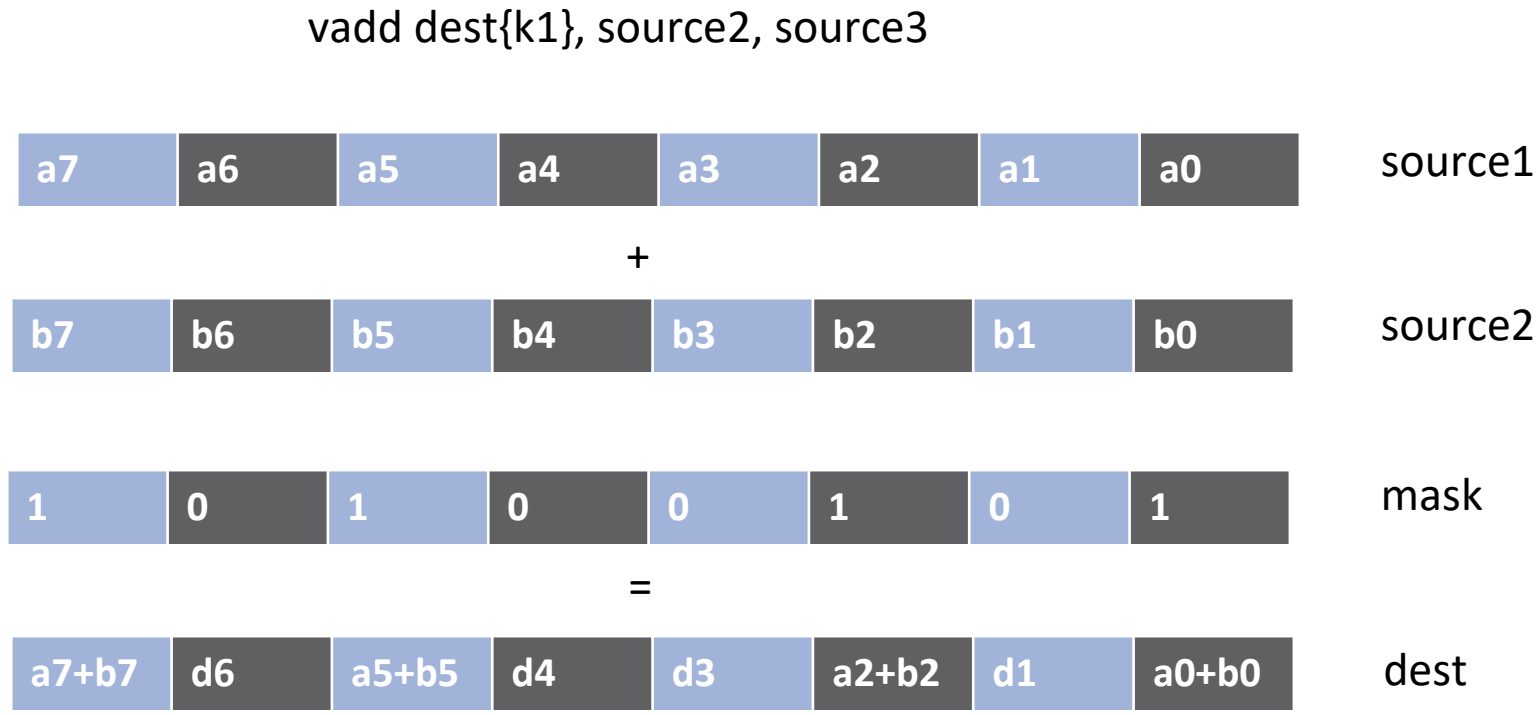
More Powerful SIMD Units

- SIMD instructions become more powerful



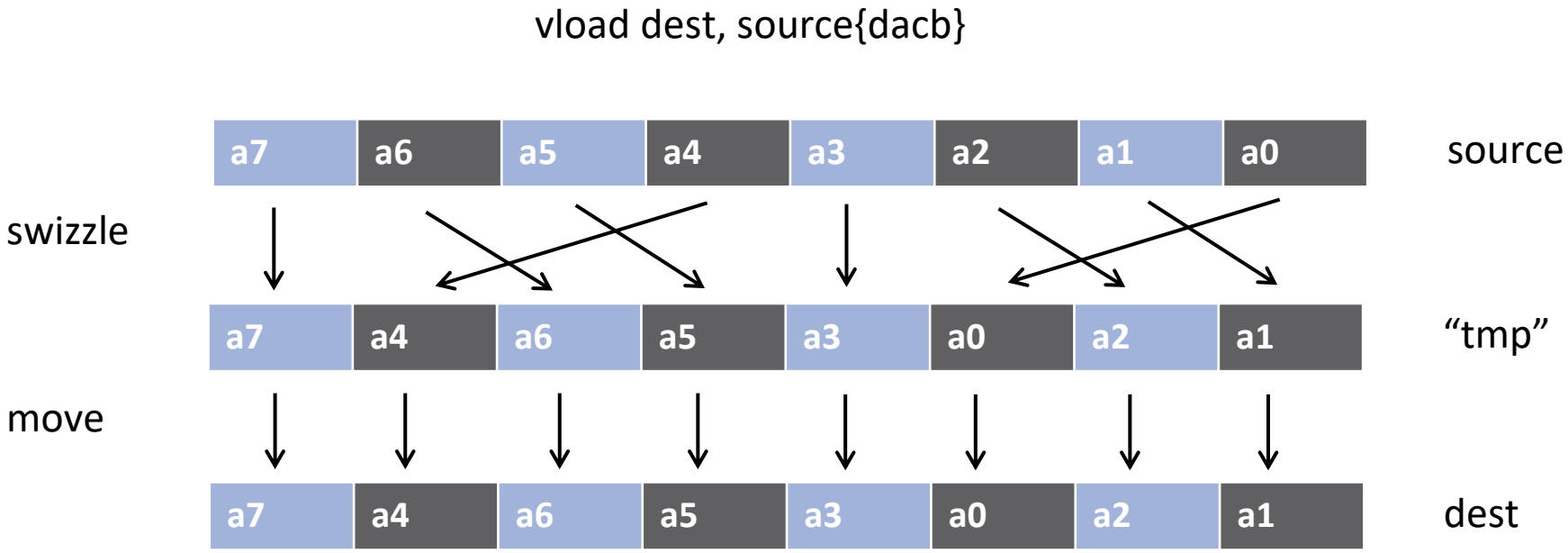
More Powerful SIMD Units

- 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
  - → 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	GCC	Intel Compiler
→ -fvectorize	-ftree-vectorize	-vec (enabled w/ -O2)
→ -Rpass=loop-.*	-ftree-loop-vectorize	-qopt-report=vec
→ -mprefer-vector-width=<*width*>	-fopt-info-vec-all	

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];
    }
}
```

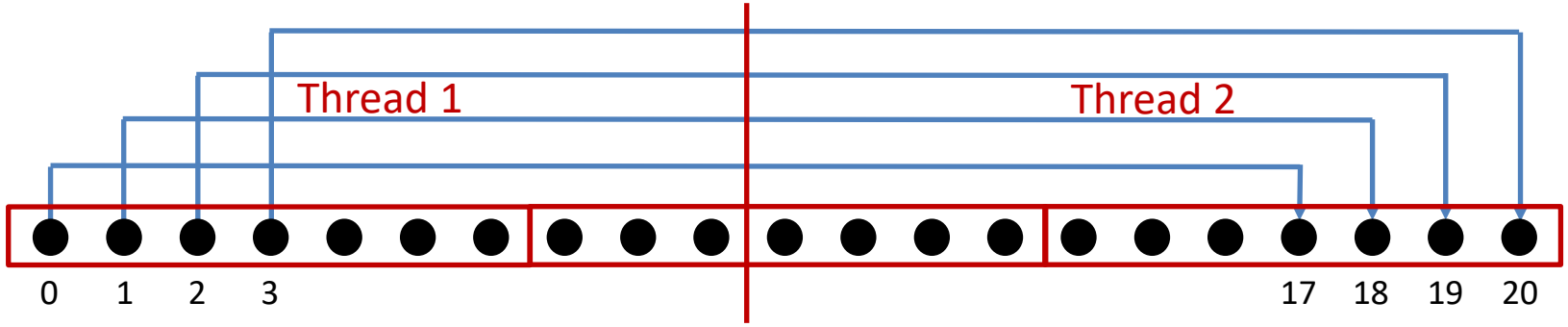
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- 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];
}   }
```



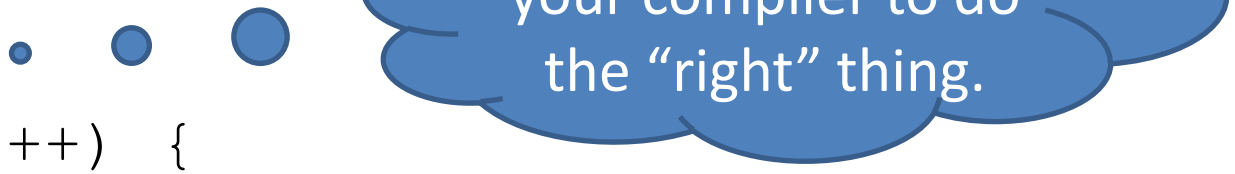
→ Parallelization: no
(except for very specific loop schedules)

→ 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] + ...;
}
```



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++)

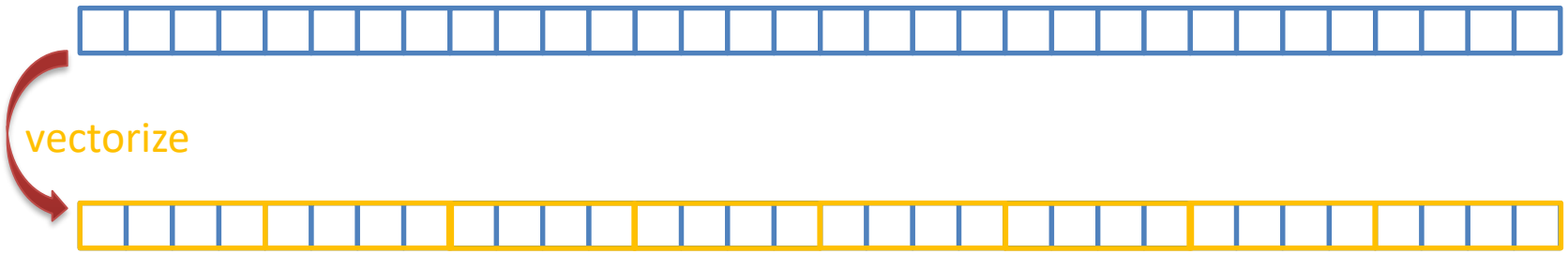
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

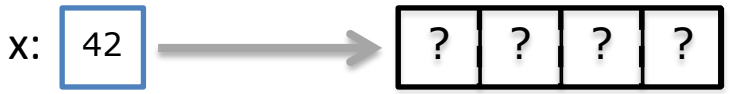
Example

```
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;
}
```

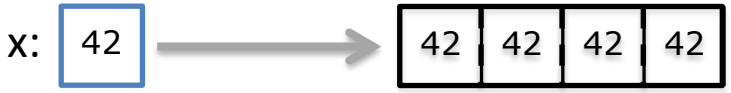


Data Sharing Clauses

- `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
- `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++)

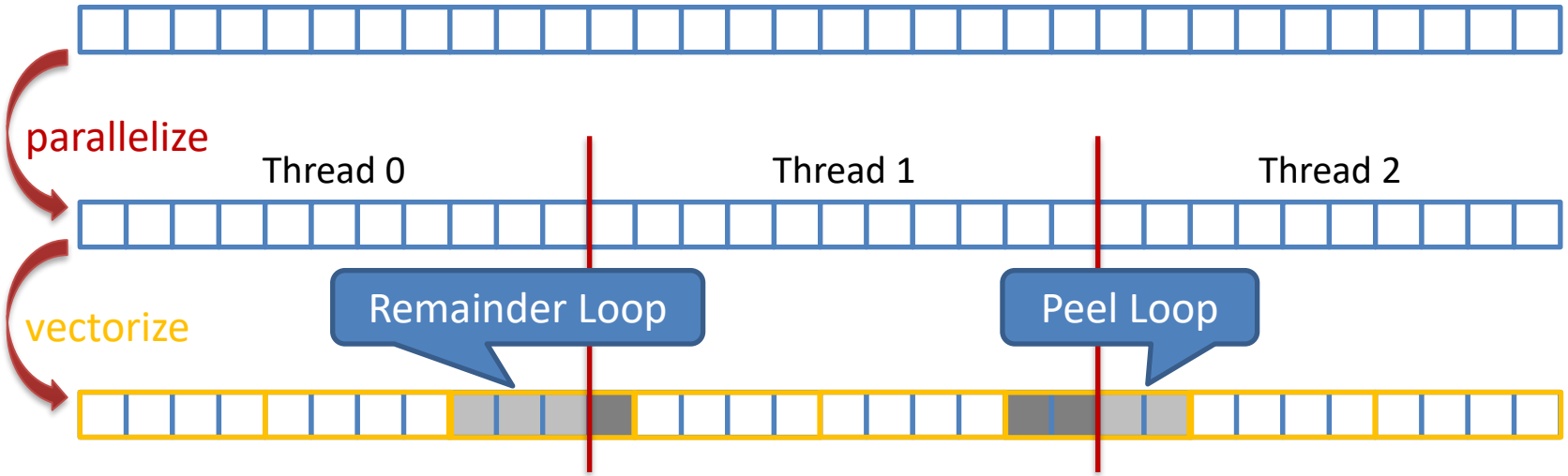
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

```
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…

```
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

```
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

```
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

```
#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
- `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])`

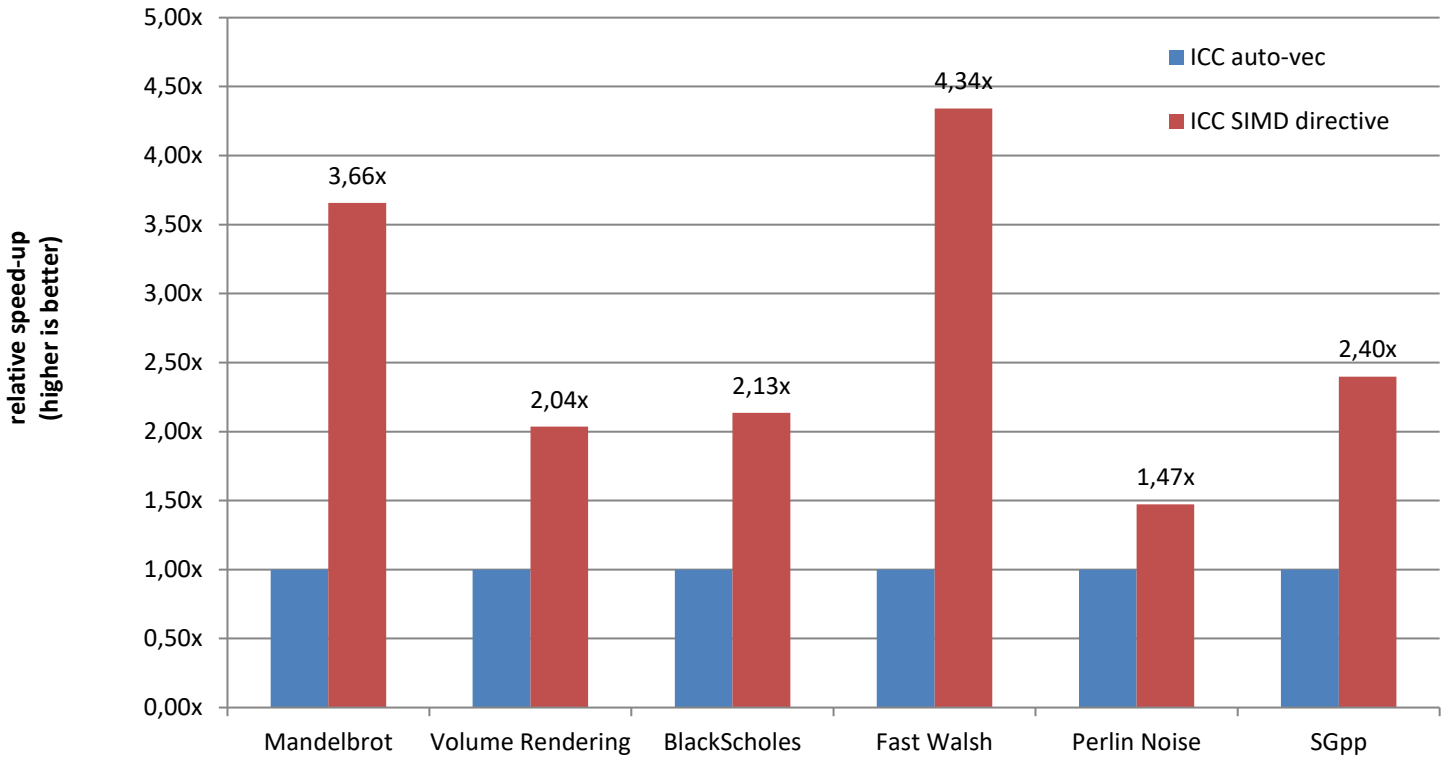
inbranch & notinbranch

```
#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]);
}
```

```
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

Using OpenMP Compilers

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

Production Compilers w/ OpenMP Support

- GCC
- clang/LLVM
- Intel Classic and Next-gen Compilers
- AOCC, AOMP, ROCmCC
- IBM XL
- … and many more

- See https://www.openmp.org/resources/openmp-compilers-tools/ for a list

Compiling OpenMP

- Enable OpenMP via the compiler's command-line switches
  - → AOCC, AOCL, ROCmCC: `-fopenmp`
  - → GCC: `-fopenmp`
  - → clang: `-fopenmp`
  - → Intel: `-fopenmp` or `–qopenmp` (classic) or `–fiopenmp` (next-gen)
  - → IBM XL: -qsmp=omp
- Switches have to be passed to both compiler and linker:

```
$ gcc [...] -fopenmp -o matmul.o -c matmul.c
$ gcc [...] -fopenmp -o matmul matmul.o
$./matmul 1024
Sum of matrix (serial):   134217728.000000, wall time 0.413975, speed-up 1.00
Sum of matrix (parallel): 134217728.000000, wall time 0.092162, speed-up 4.49
```

Programming OpenMP

Hands-on Exercises

Christian Terboven

Michael Klemm

RWTH AACHEN UNIVERSITY

Webinar Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - → Download: `git clone https://github.com/cterboven/OpenMP-tutorial-CSC.git`
  - → Build: `make`
  - → You can then find the compiled code in the “bin” folder to run it
  - → We use the GCC compiler mostly, some examples require Intel’s Math Kernel Library

- 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

- Please, feel free to also work on your own code; this is your change to talk with us via the chat to discuss your OpenMP needs, findings, problems, and everything else.