

Programming OpenMP

Christian Terboven
Michael Klemm



Agenda

	Day 1	Day 2	Day 3
09:00-10:30 CET		Tasking 1 <ul style="list-style-type: none">Tasking IntroLab 1	GPUs <ul style="list-style-type: none">OpenMP for Compute Accelerators
10:45-12:15 CET		Tasking 2 <ul style="list-style-type: none">TaskloopDependenciesCancellationLab 2	Tools for Perf. and Correctness <ul style="list-style-type: none">VI-HPS Tools for PerformanceVI-HPS Tools for Correctness
13:00-14:45 CET	Introduction to OpenMP	Host Perf.: SIMD <ul style="list-style-type: none">VectorisationLab 3	Misc. OpenMP 5.0 Features <ul style="list-style-type: none">DOACROSS Loops
15:00-16:00 CET	Hands-on: Introduction to OpenMP	Host Perf.: NUMA <ul style="list-style-type: none">Memory AccessTask AffinityMemory ManagementLab 4	Roadmap / Outlook <ul style="list-style-type: none">Open DiscussionOpenMP 5.0 and beyond <p>End: approx. 16:30 CET</p>

Programming OpenMP

An Overview Of OpenMP

Christian Terboven

Michael Klemm



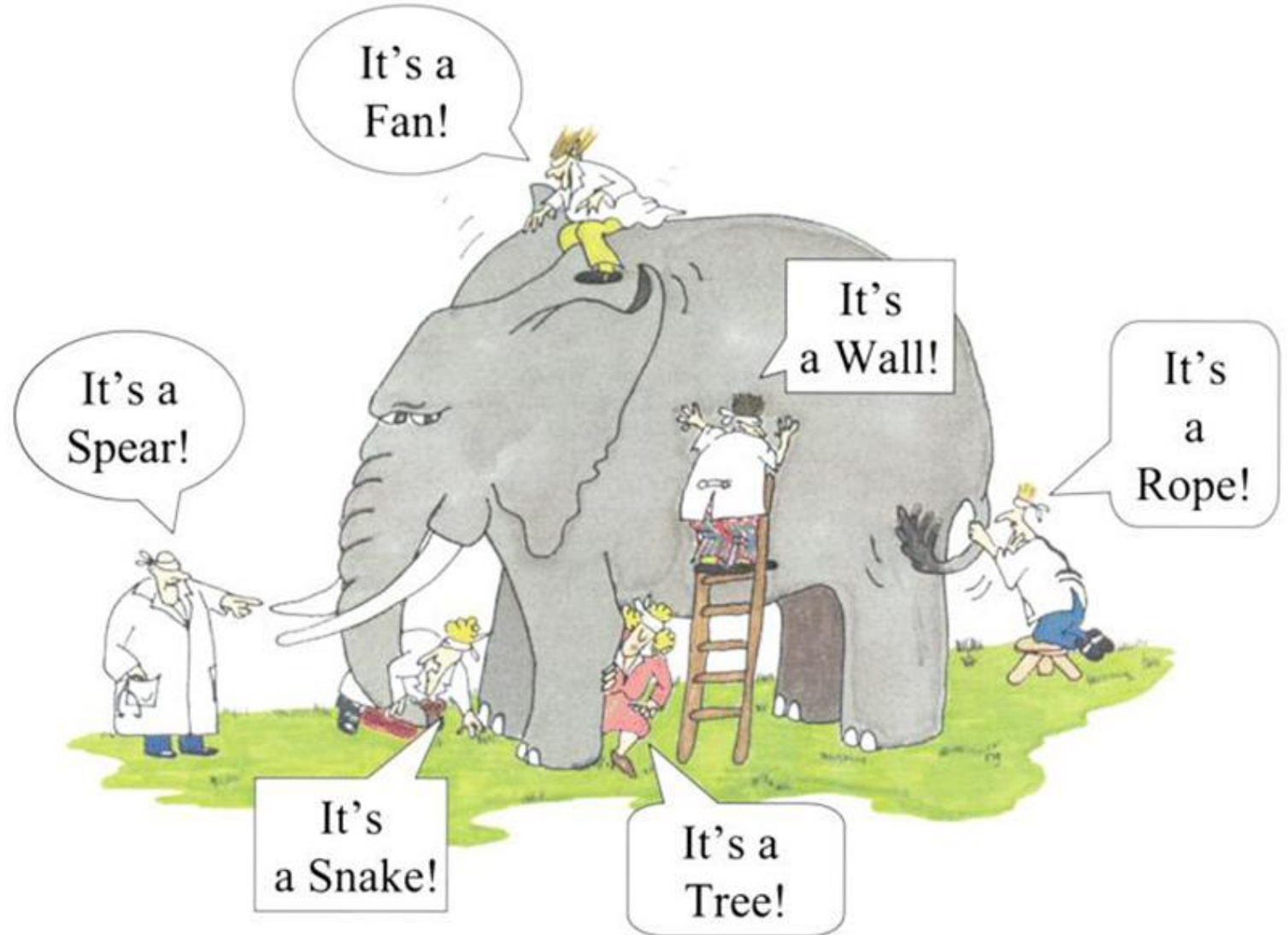
History

- 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



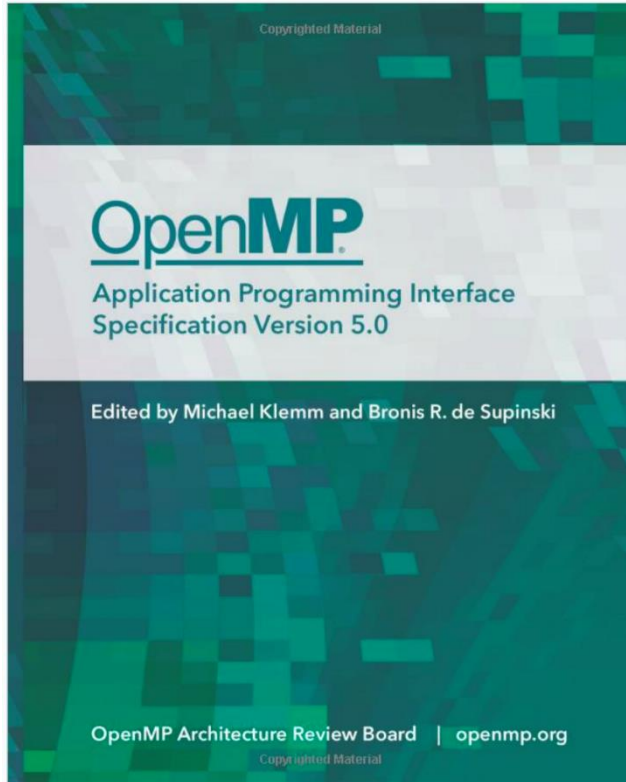
What is OpenMP?

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

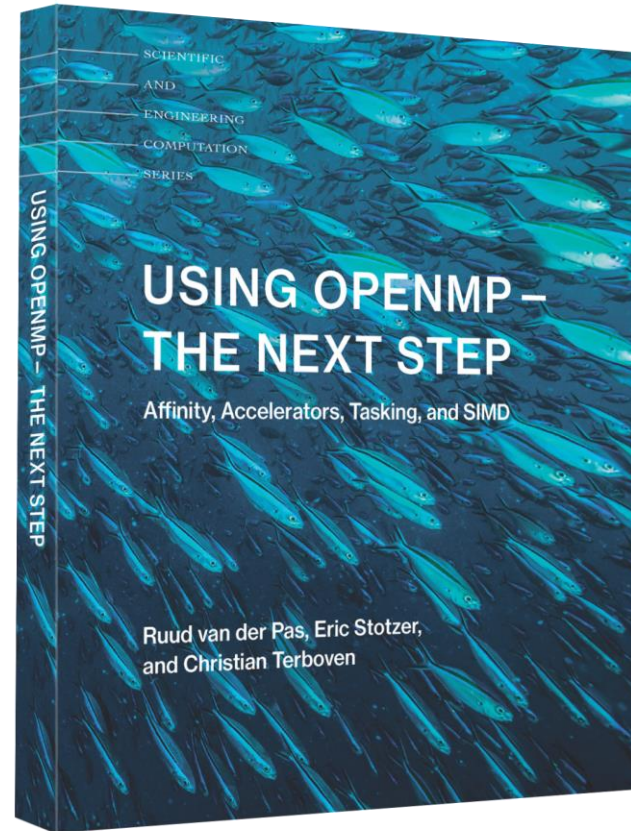


[illegible]

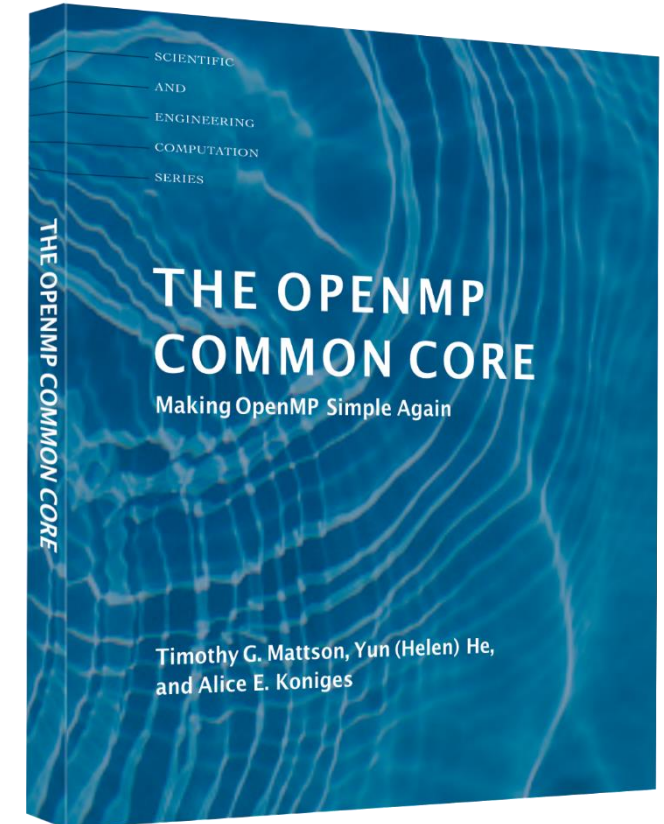
Recent Books About OpenMP



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

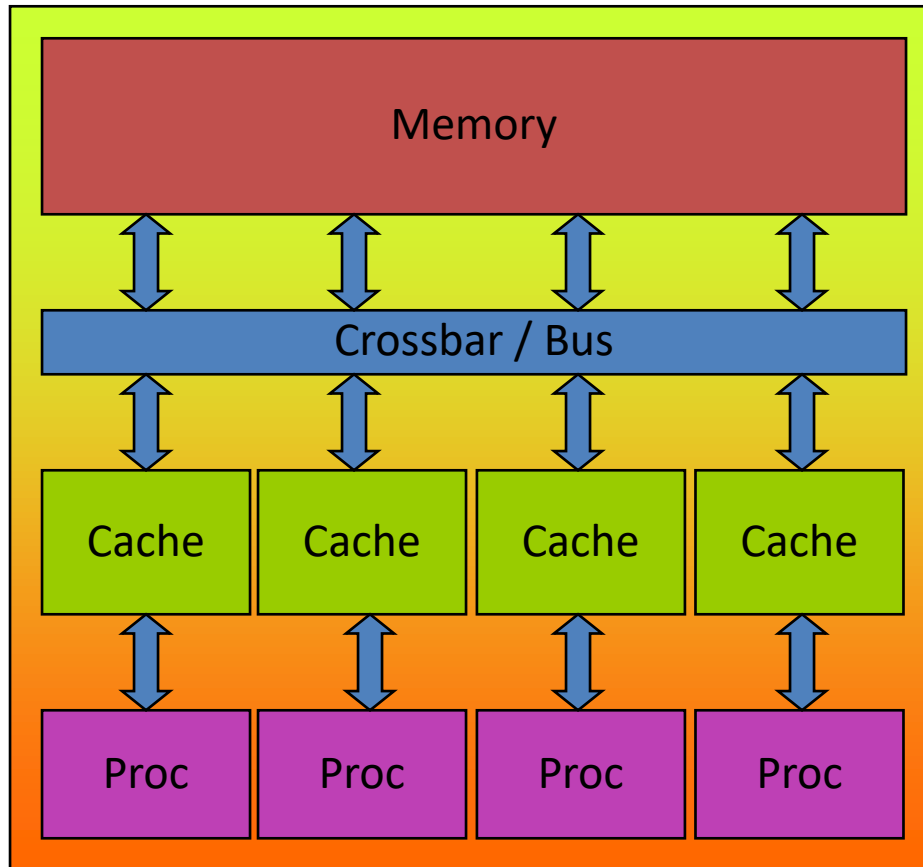
Christian Terboven

Michael Klemm



OpenMP's machine model

- OpenMP: Shared-Memory Parallel Programming Model.



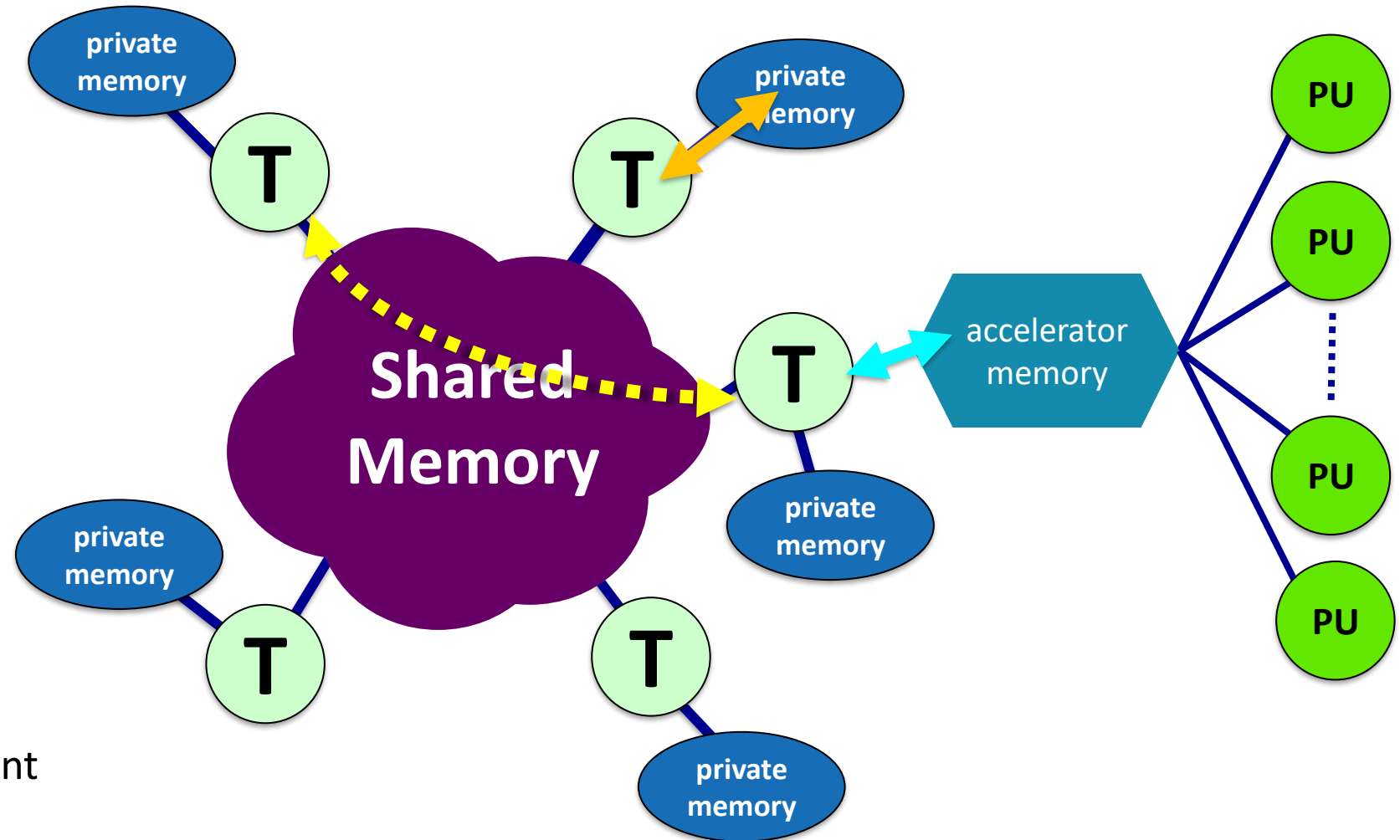
All processors/cores access a shared main memory.

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

Parallelization in OpenMP employs multiple threads.

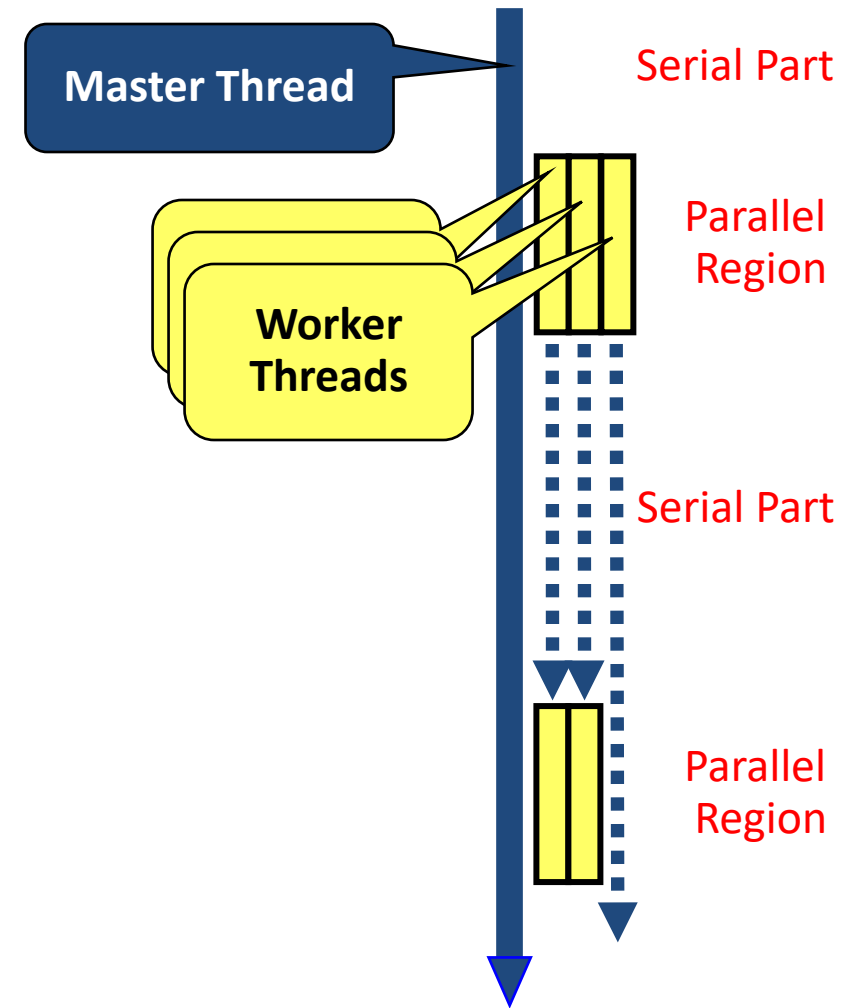
The OpenMP Memory Model

- 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



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

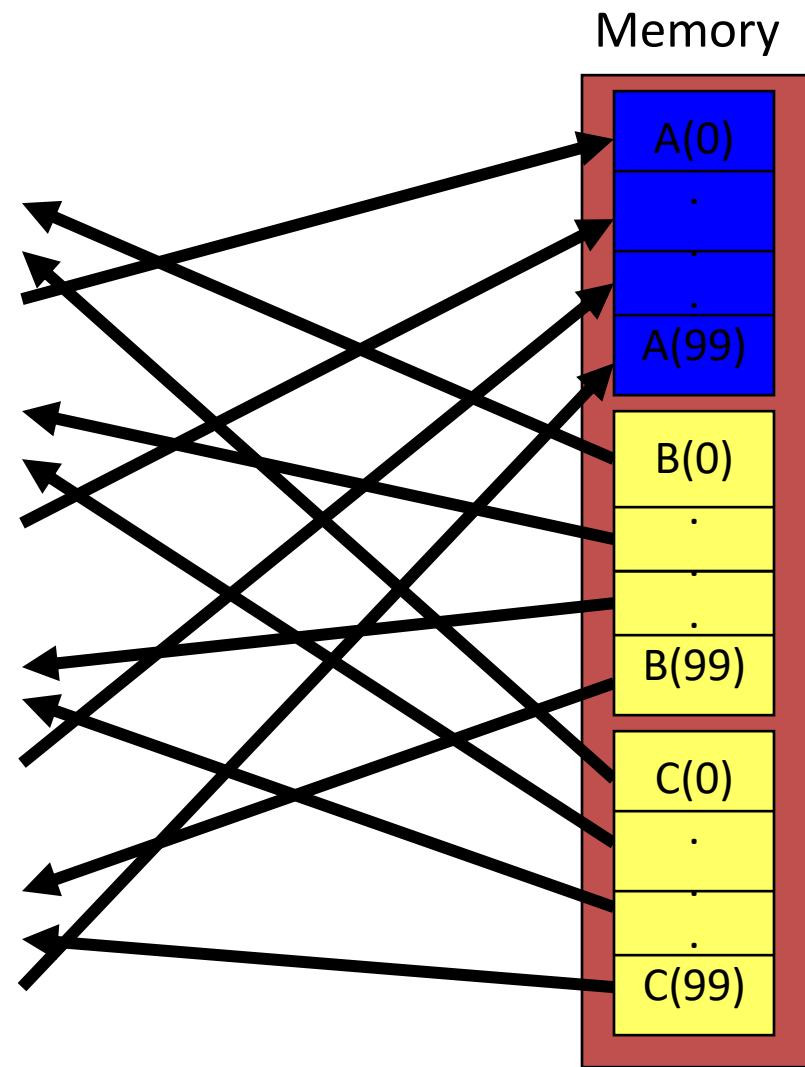
```
do i = 25, 49  
  a(i) = b(i) + c(i)  
end do
```

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 one 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)`.

■ Static Schedule

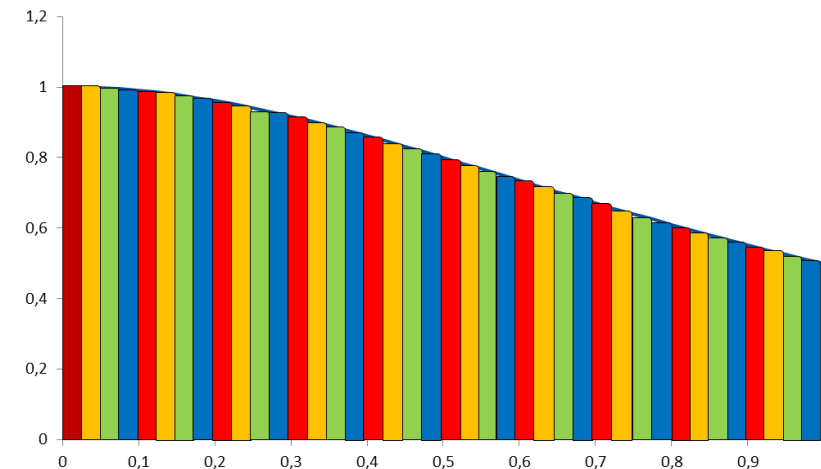
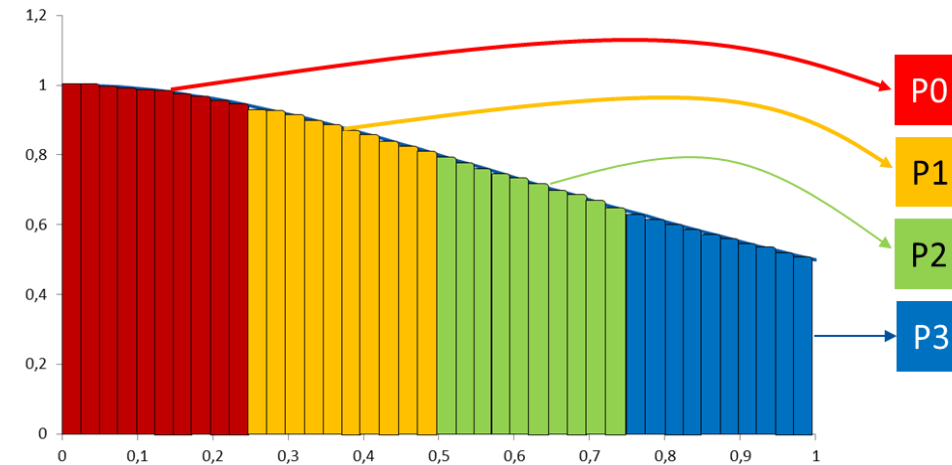
- `schedule(static [, chunk])`
- Decomposition
depending on chunksize
- Equal parts of size 'chunksize'
distributed in round-robin
fashion

■ 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



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)

C/C++

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

Fortran

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

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

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
```

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



do i = 0, 24
s₁ = s₁ + a(i)
end do
s = s + s₁

do i = 25, 49
s₂ = s₂ + a(i)
end do
s = s + s₂

do i = 50, 74
s₃ = s₃ + a(i)
end do
s = s + s₃

do i = 75, 99
s₄ = s₄ + a(i)
end do
s = s + s₄

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)

PI

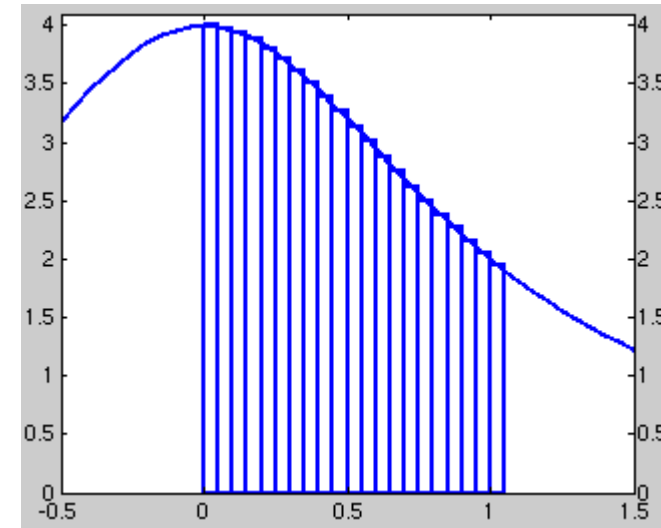
Example: Pi (1/2)

```
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;
}
```

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



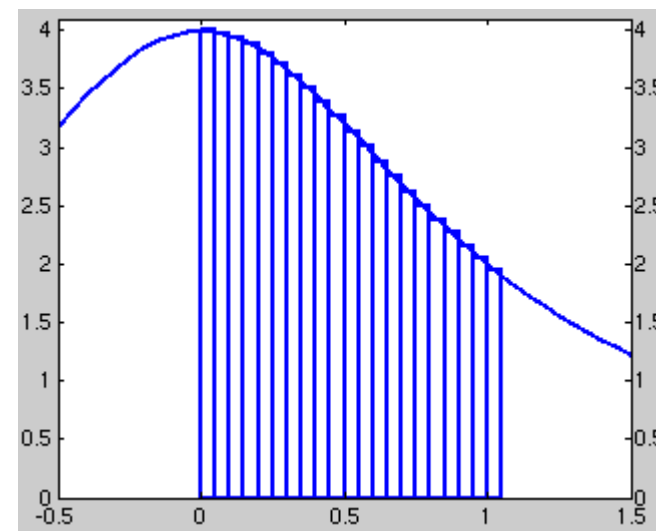
Example: Pi (2/2)

```
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



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> ); ...;  
}
```

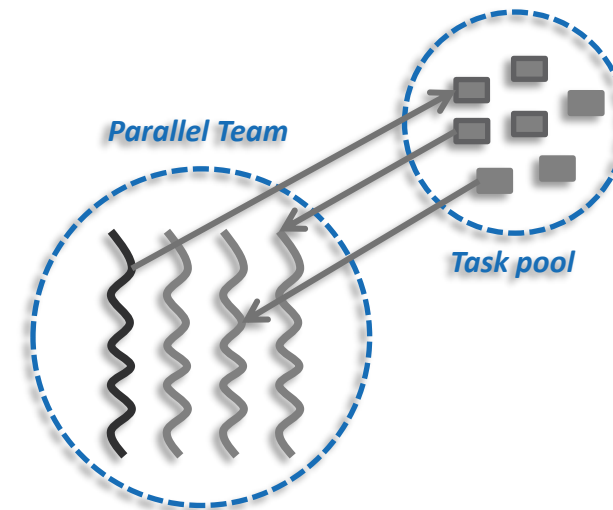
- Several scenarios are possible:

→ 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 }
```

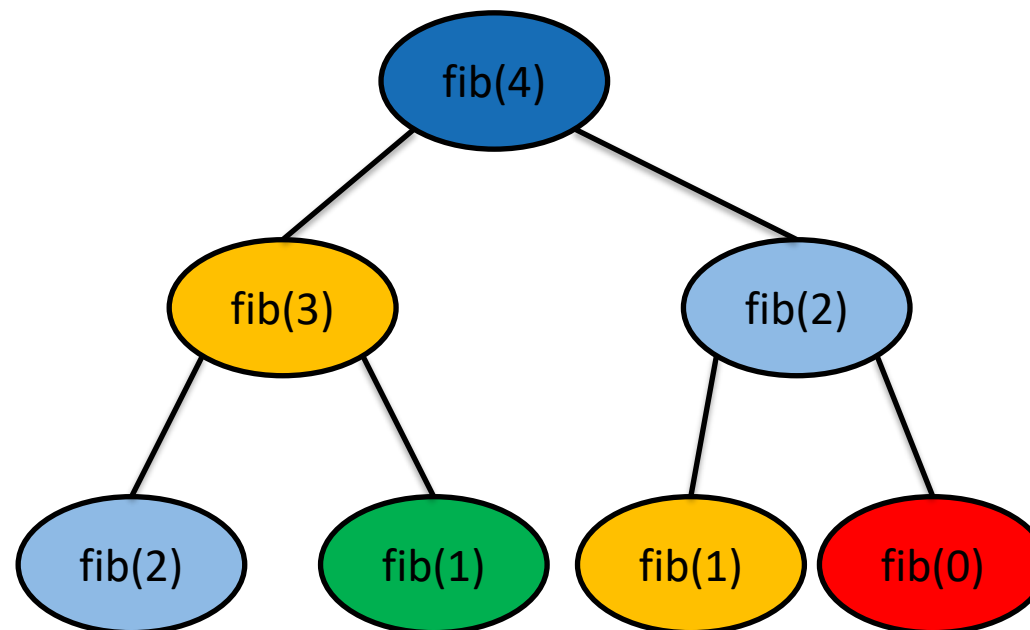
Fibonacci Numbers (in a Stupid Way 😊)

```
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 }
```

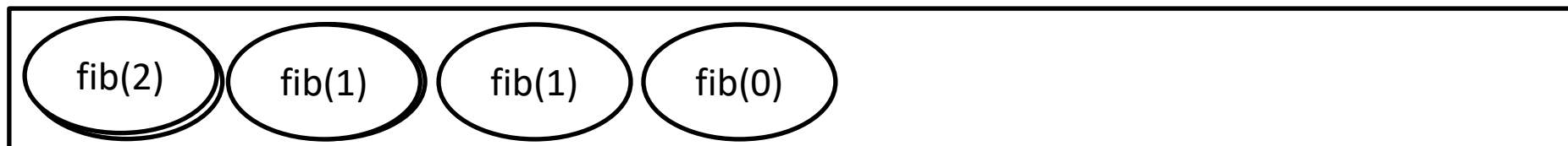
```
14 int fib(int n) {  
15     if (n < 2) return n;  
16     int x, y;  
17     #pragma omp task shared(x)  
18     {  
19         x = fib(n - 1);  
20     }  
21     #pragma omp task shared(y)  
22     {  
23         y = fib(n - 2);  
24     }  
25     #pragma omp taskwait  
26     return x+y;  
27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

- 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.

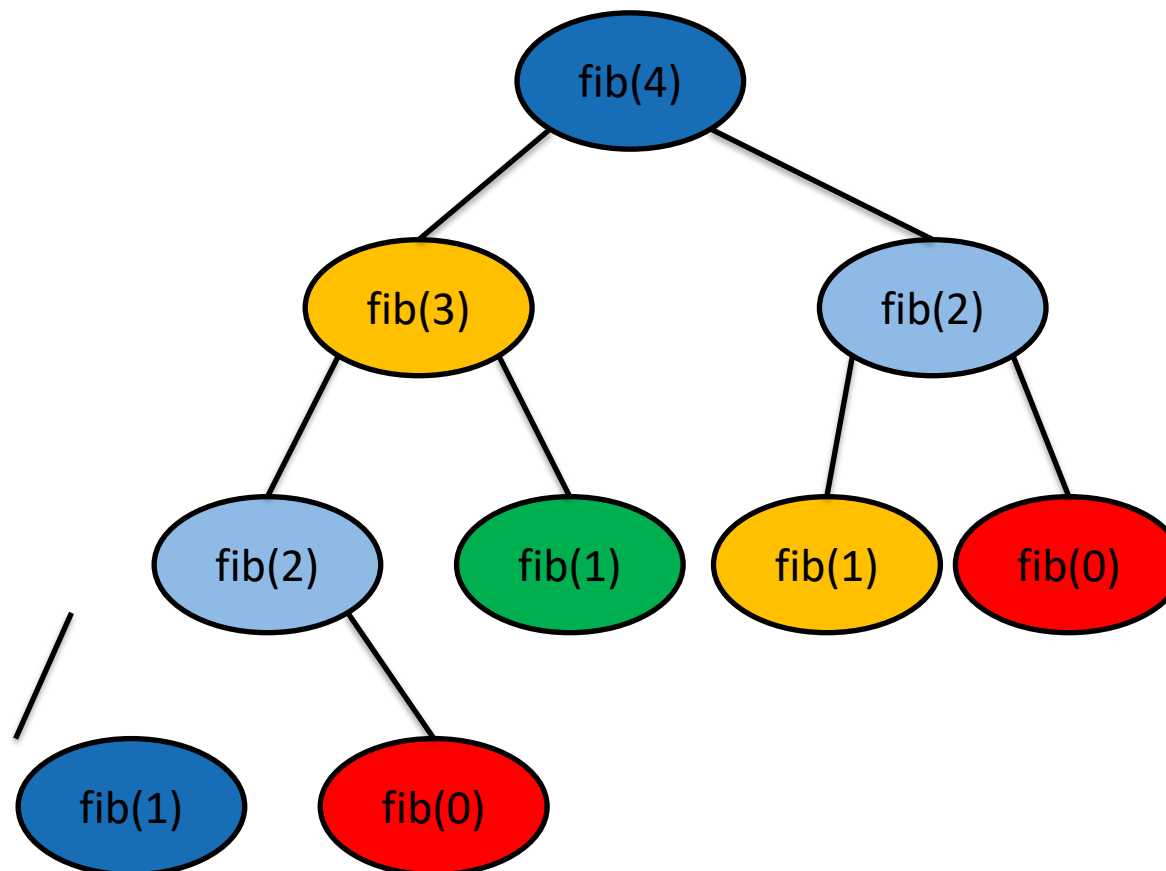
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



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
- ...



Programming OpenMP

Using OpenMP Compilers

Christian Terboven

Michael Klemm



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
 - GCC: `-fopenmp`
 - clang: `-fopenmp`
 - Intel: `-fopenmp` or `-qopenmp` (classic) or `-fiopenmp` (next-gen)
 - AOCC, AOCL, ROCmCC: `-fopenmp`
 - 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



- 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-PRACE.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