

Programming OpenMP

Christian Terboven



Michael Klemm



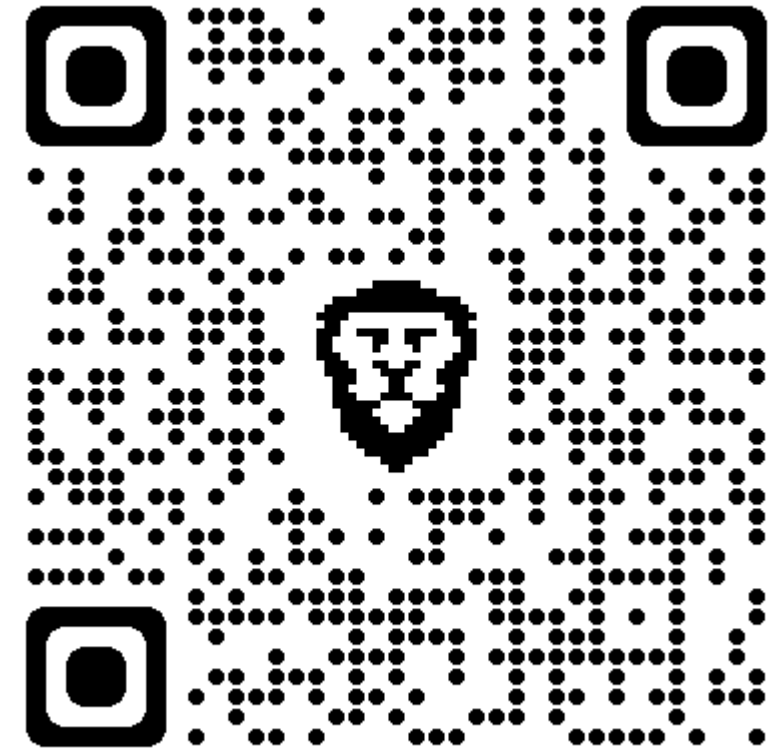
Agenda (tentative – tell us what else you need)

| | Day 1 | Day 2 | Day 3 |
|-----------------|-------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 09:00-10:30 CET | Introduction to OpenMP 1 | Tasking 1 <ul style="list-style-type: none">• Tasking Intro• Lab 1 | GPUs <ul style="list-style-type: none">• OpenMP for Compute Accelerators |
| 10:45-12:15 CET | Hands-on: Introduction to OpenMP | Tasking 2 <ul style="list-style-type: none">• Taskloop• Dependencies• Cancellation• Lab 2 | Tools for Perf. and Correctness <ul style="list-style-type: none">• VI-HPS Tools for Performance• VI-HPS Tools for Correctness |
| 13:00-14:45 CET | Introduction to OpenMP 2 | Host Perf.: SIMD <ul style="list-style-type: none">• Vectorisation• Lab 3 | Misc. OpenMP 5.1 Features <ul style="list-style-type: none">• DOACROSS Loops |
| 15:00-16:00 CET | Hands-on: Introduction to OpenMP If requested | Host Perf.: NUMA <ul style="list-style-type: none">• Memory Access• Task Affinity• Memory Management• Lab 4 | Roadmap / Outlook <ul style="list-style-type: none">• Open Discussion• OpenMP 5.1 and beyond |

Lab: hands-on time

Material

- You can find all on github.com:
 - Slide decks
 - Exercise tasks
 - Solutions
- <https://github.com/cterboven/OpenMP-tutorial-PRACE-2022>



Programming OpenMP

An Overview Of OpenMP

Christian Terboven

Michael Klemm



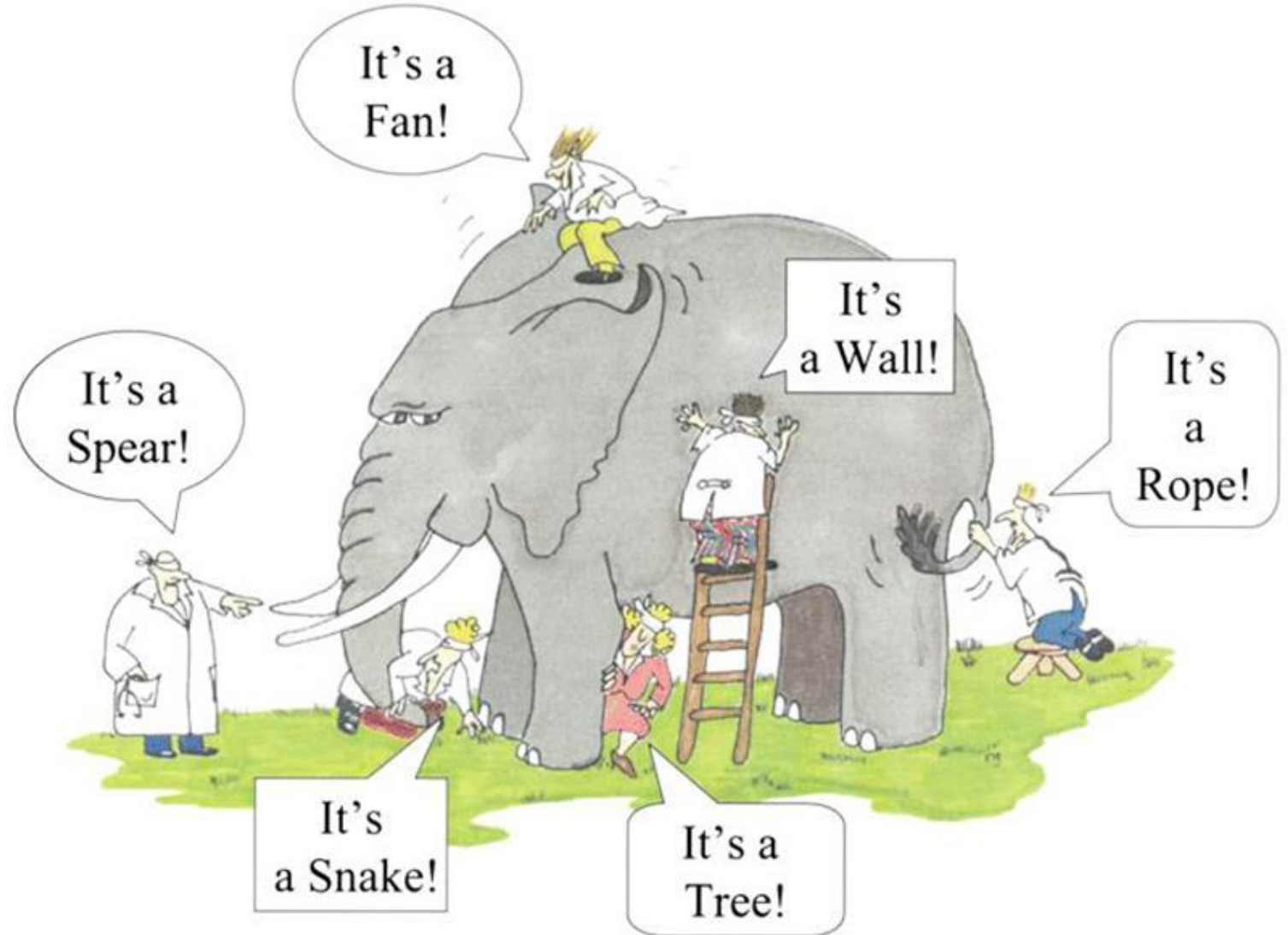
History

- 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
- 11/2021: OpenMP 5.2



What is OpenMP?

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

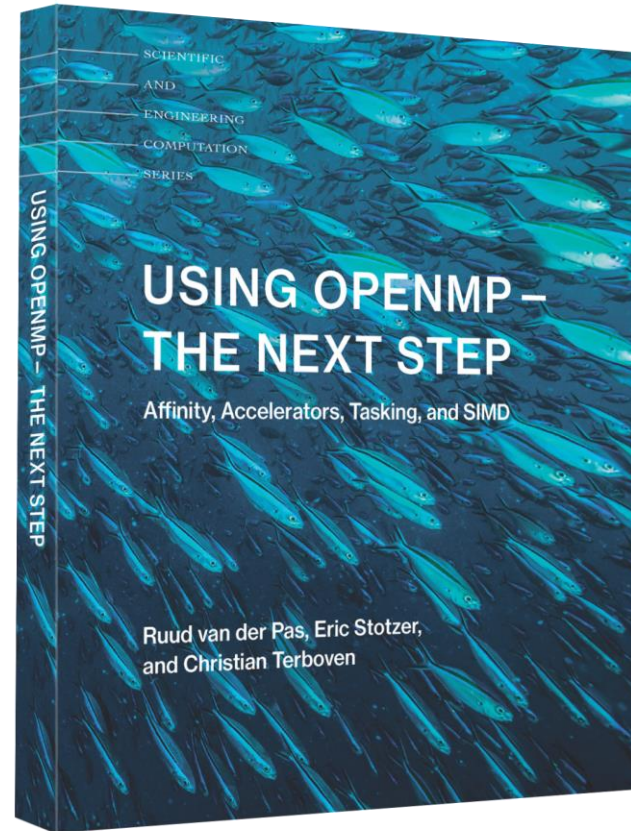


[illegible]

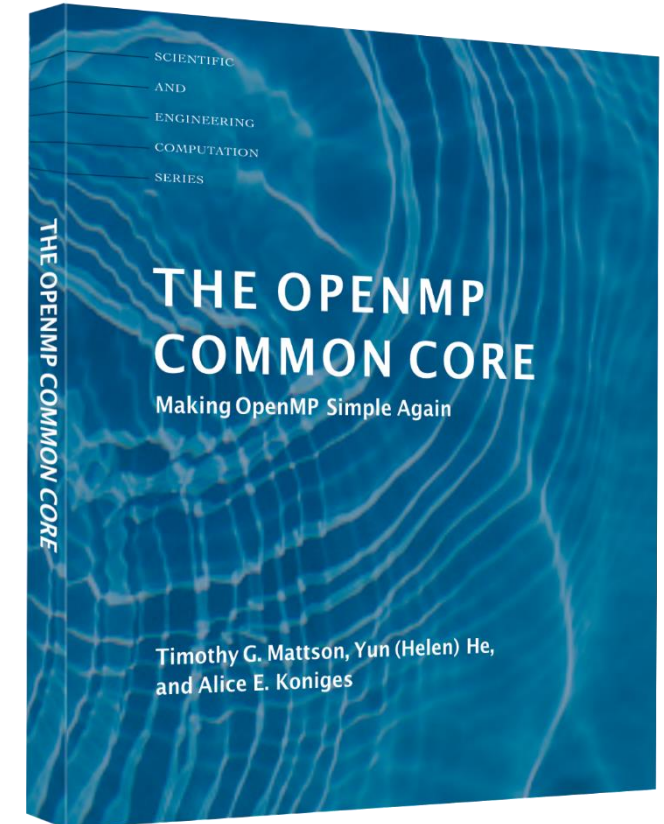
Recent Books About OpenMP



A printed copy of the 5.1 specifications, 2020



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

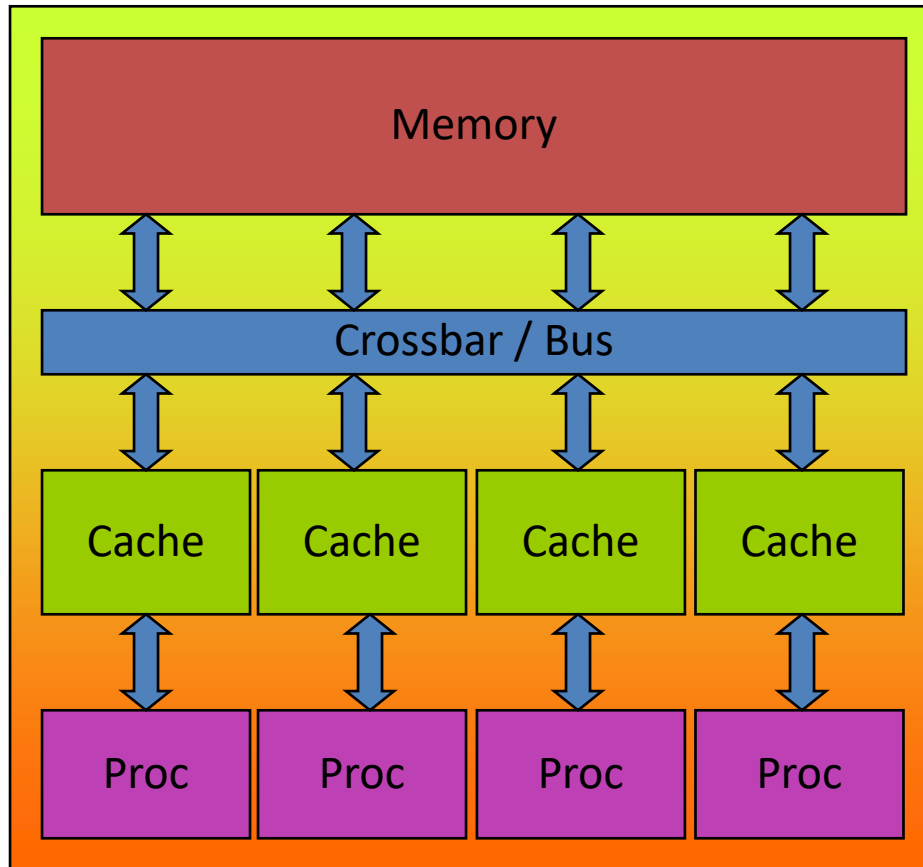
Christian Terboven

Michael Klemm



OpenMP's machine model

- OpenMP: Shared-Memory Parallel Programming Model.



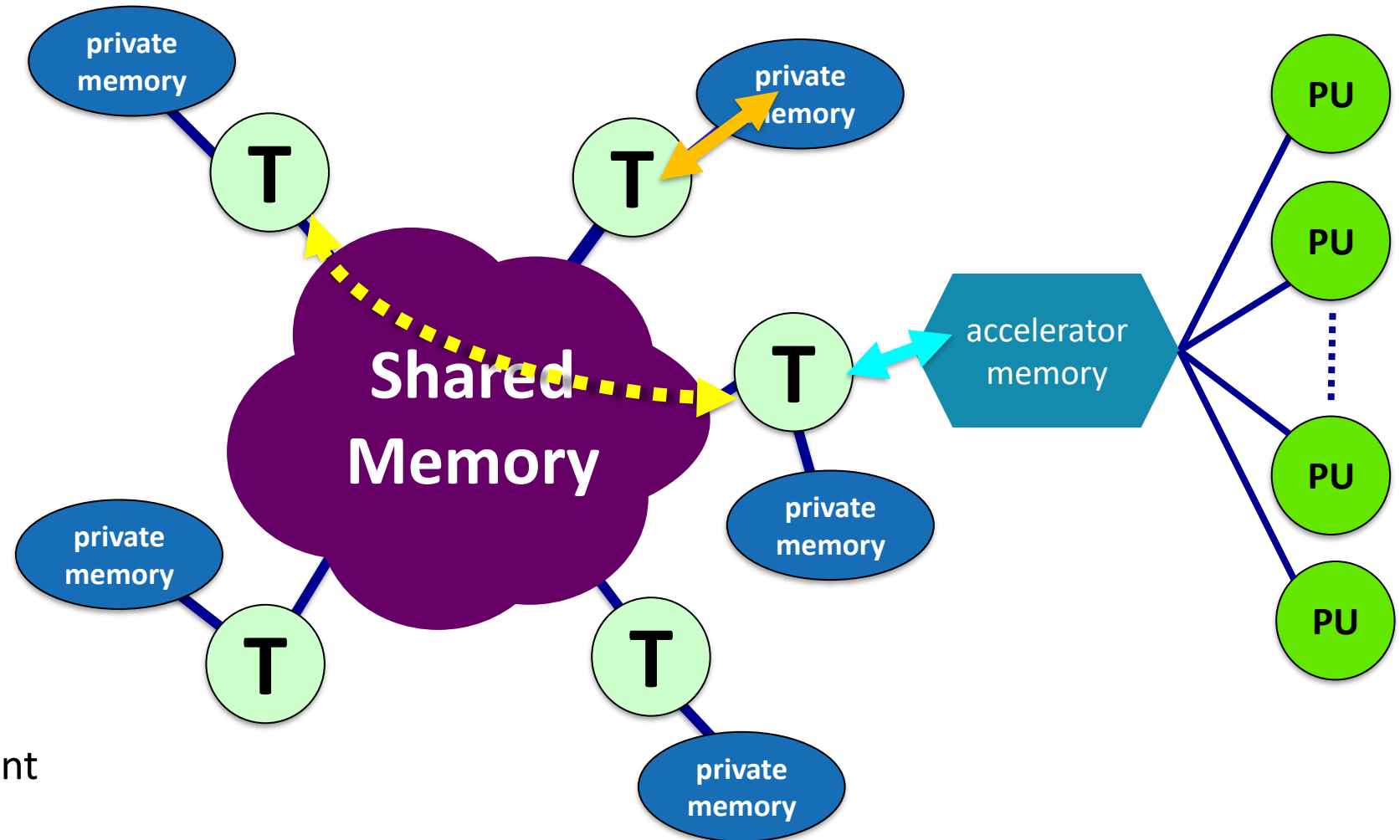
All processors/cores access a shared main memory.

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

Parallelization in OpenMP employs multiple threads.

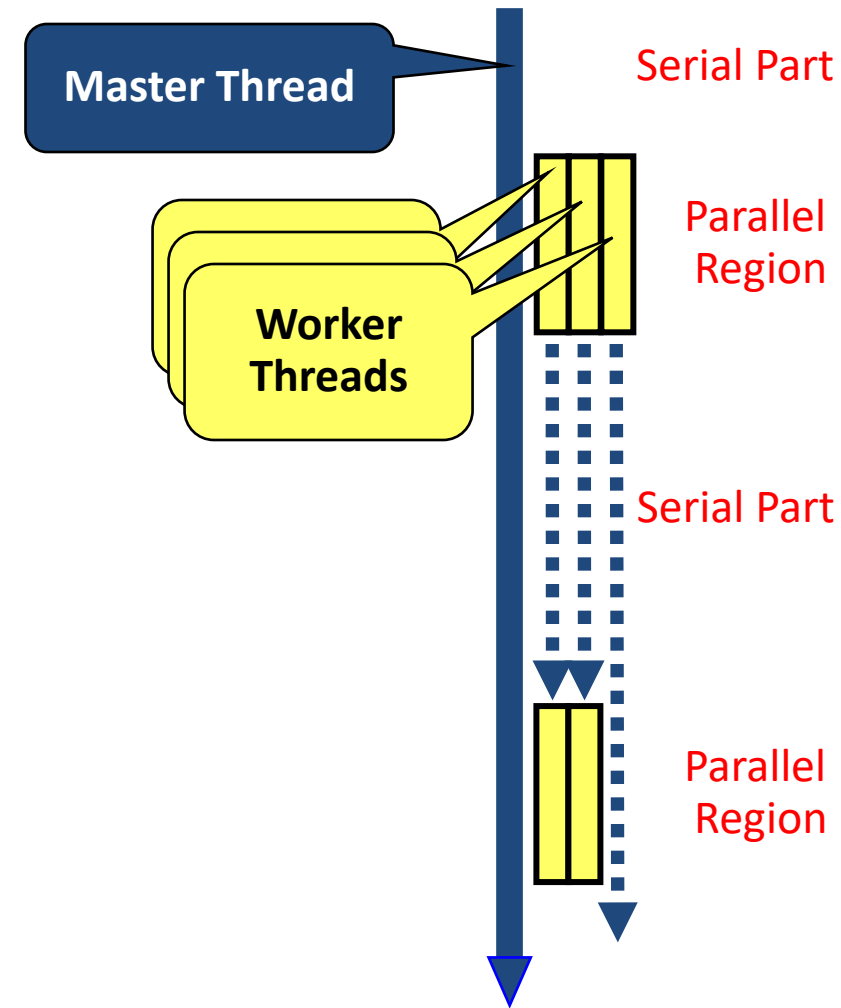
The OpenMP Memory Model

- 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
```

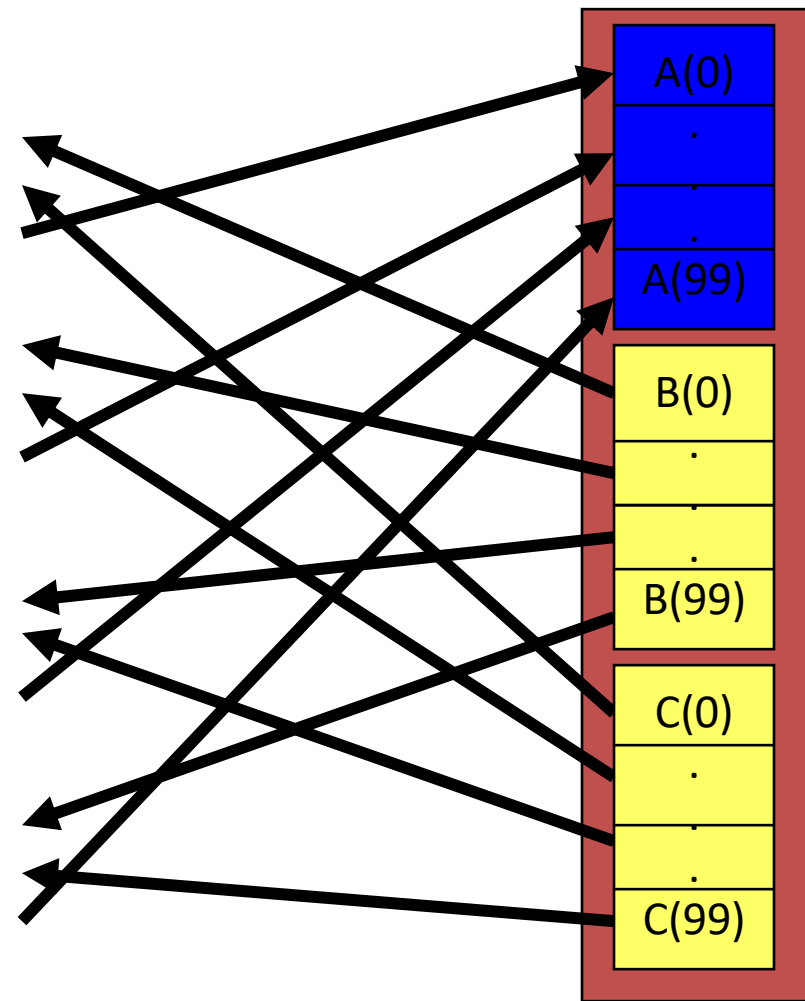
Thread 3

```
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
```

Memory



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

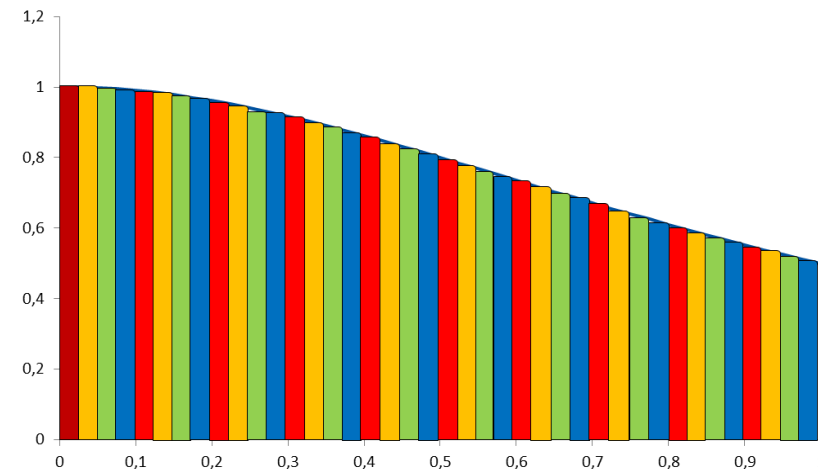
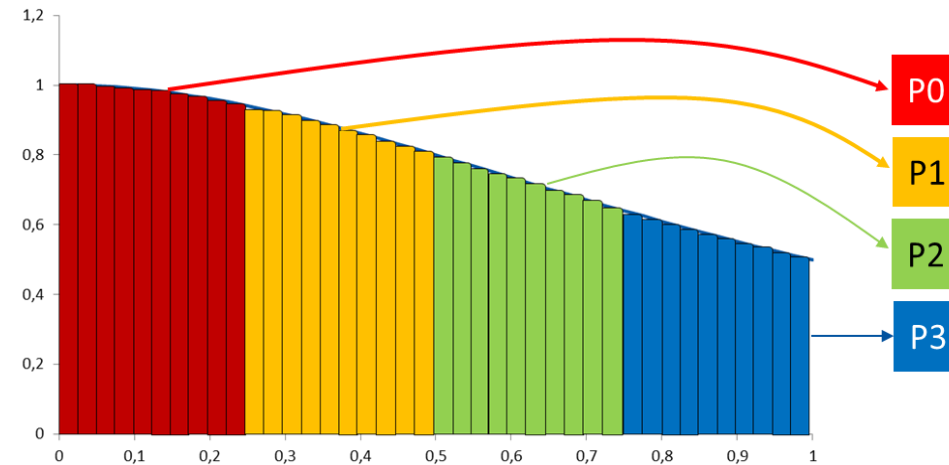
- `schedule(static [, chunk])`
- Decomposition
depending on chunksize
- Equal parts of size 'chunksize'
distributed in round-robin
fashion

■ 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


```
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;
}
```

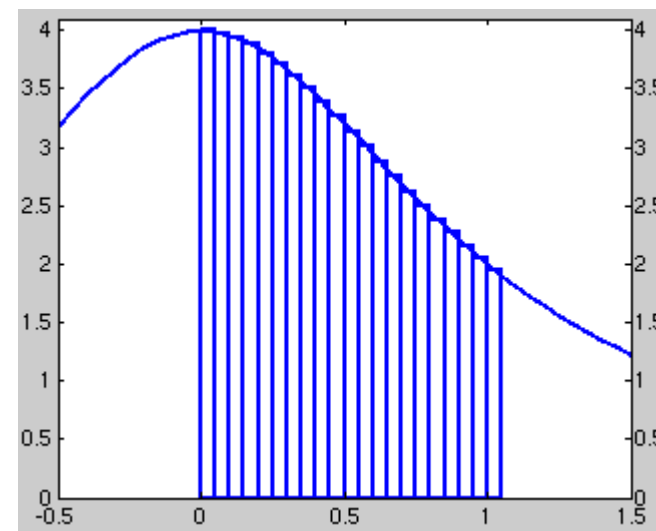
Example: Pi (2/2)

```
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;
}
```

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



PI

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-2022.git`
 - Build: `make` (in the corresponding subdirectories)
 - 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
- Also provided: basic exercises in the [openmp-simple-exercises.tar](#) archive
 - Instructions contained in the archive: [Exercises_OMP_2021.pdf](#)

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> ); ...;  
}
```

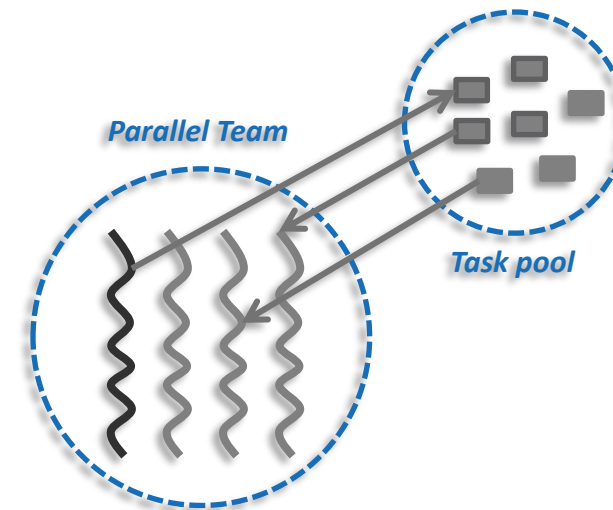
- Several scenarios are possible:

→ 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 }
```

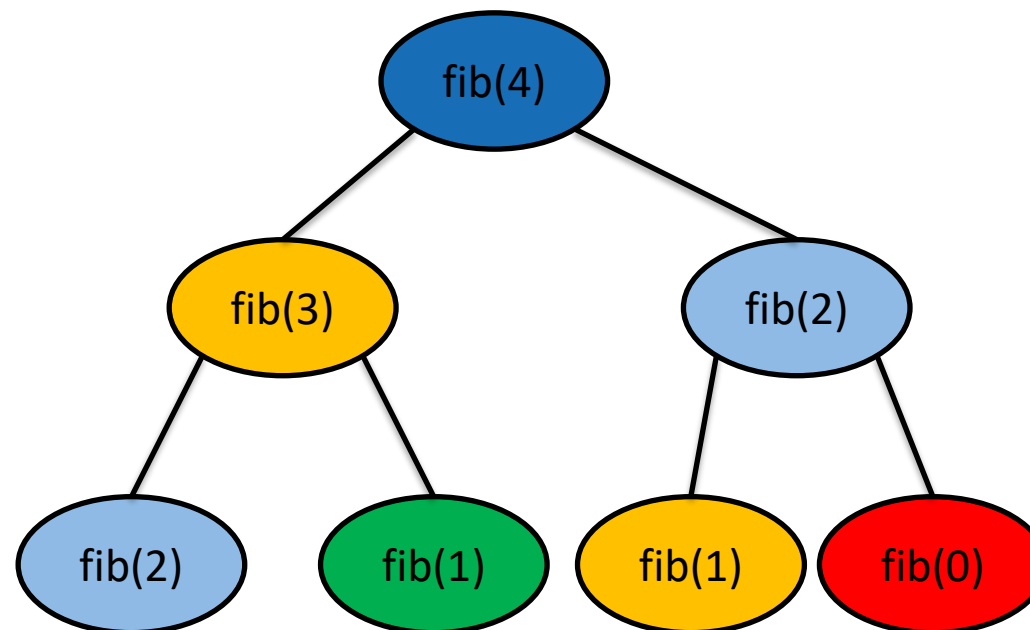
Fibonacci Numbers (in a Stupid Way 😊)

```
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 }
```

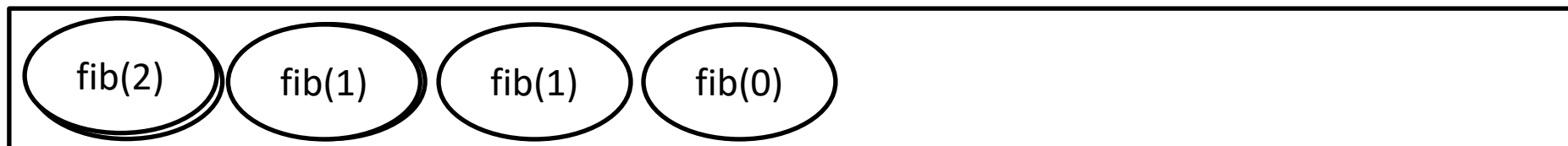
```
14 int fib(int n) {  
15     if (n < 2) return n;  
16     int x, y;  
17     #pragma omp task shared(x)  
18     {  
19         x = fib(n - 1);  
20     }  
21     #pragma omp task shared(y)  
22     {  
23         y = fib(n - 2);  
24     }  
25     #pragma omp taskwait  
26     return x+y;  
27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

- 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.

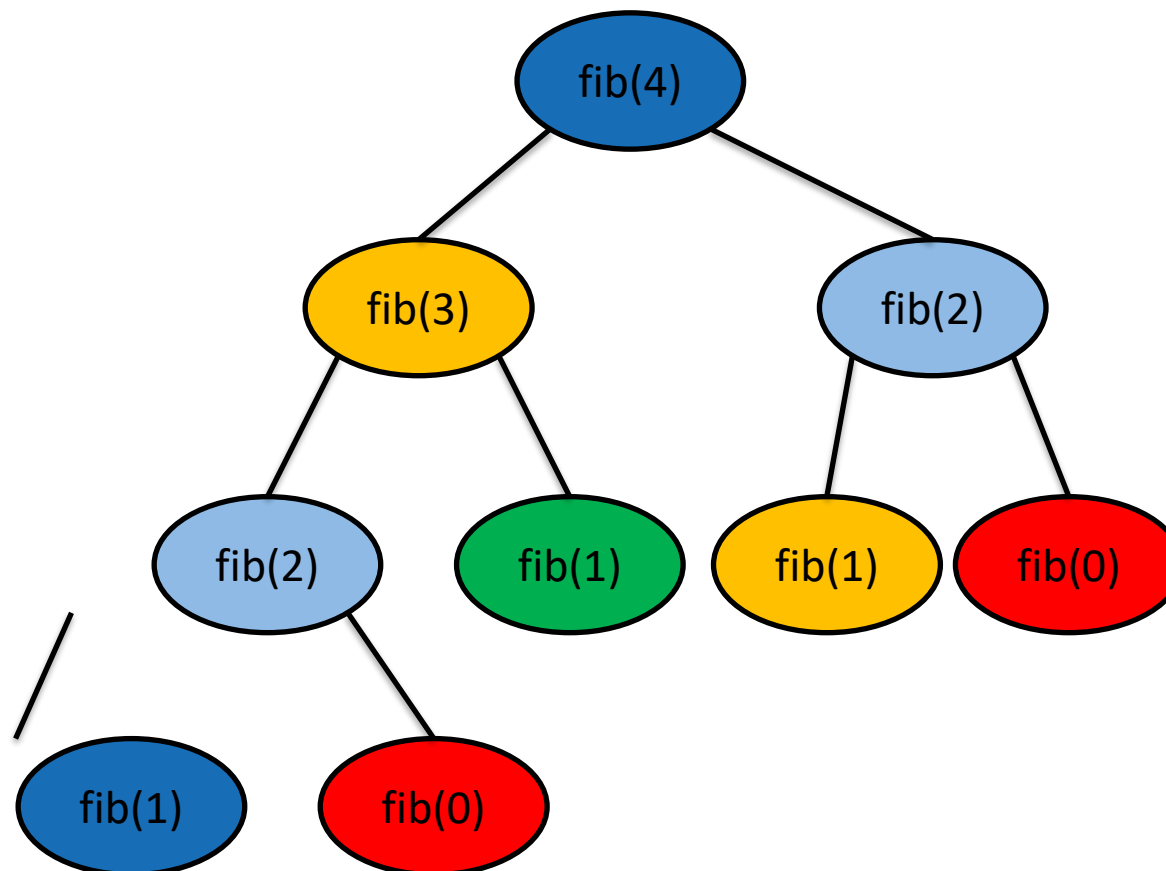
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



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

Hands-on Exercises

Christian Terboven

Michael Klemm



Solution of Homework Assignments

Example: sin-cos

```
double do_some_computation(int i) {
    double t = 0.0;
    for (int j = 0; j < i*i; j++) {
        t += sin((double)j) * cos((double)j);
    }
    return t;
}

int main(int argc, char* argv[]) {
    const int dimension = 500;
    int i;
    double result = 0.0;
    double t1 = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic) reduction(+:result)
    for (i = 0; i < dimension; i++) {
        result += do_some_computation(i);
    }
    double t2 = omp_get_wtime();
    printf("Computation took %.3lf seconds.\n", t2 - t1);
    printf("Result is %.3lf.\n", result);
    return 0;
}
```

Example: matmul

```
void matmul_seq(double * C, double * A, double * B, size_t n) { ... }

void matmul_par(double * C, double * A, double * B, size_t n) {
    #pragma omp parallel for shared(A,B,C) firstprivate(n) \
        schedule(static) // collapse(2)
    for (size_t i = 0; i < n; ++i) {
        for (size_t k = 0; k < n; ++k) {
            for (size_t j = 0; j < n; ++j) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
        }
    }
}


void init_mat(double * C, double * A, double * B, size_t n) { ... }

void dump_mat(double * mtx, size_t n) { ... }
double sum_mat(double * mtx, size_t n) { ... }

int main(int argc, char *argv[]) { ... }
```

Example: cholesky

```
void cholesky(int ts, int nt, double* Ah[nt][nt]) {  
    for (int k = 0; k < nt; k++) {  
        LAPACKE_dpotrf(LAPACK_COL_MAJOR, 'L', ts, Ah[k][k], ts);  
  
        #pragma omp parallel for  
        for (int i = k + 1; i < nt; i++) {  
            cblas_dtrsm(CblasColMajor, CblasRight, CblasLower, CblasTrans,  
                        CblasNonUnit, ts, ts, 1.0, Ah[k][k], ts, Ah[k][i], ts);  
        }  
  
        #pragma omp parallel for  
        for (int i = k + 1; i < nt; i++) {  
            for (int j = k + 1; j < i; j++) {  
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasTrans, ts, ts, ts, -1.0,  
                            Ah[k][i], ts, Ah[k][j], ts, 1.0, Ah[j][i], ts);  
            }  
            cblas_dsyrk(CblasColMajor, CblasLower, CblasNoTrans, ts, ts, -1.0,  
                        Ah[k][i], ts, 1.0, Ah[i][i], ts);  
        }  
    }  
}
```



Blocked matrix
w/ block size *ts*

Advanced OpenMP Tutorial

Misc Stuff

Christian Terboven

Michael Klemm

The logo for RWTH Aachen University, with "RWTHAACHEN" in blue and "UNIVERSITY" in a lighter blue below it.The OpenMP logo, consisting of the text "OpenMP" in a teal color, with a horizontal line underneath the "Open" part.

Masked construct

Masked construct in OpenMP 5.1 / 1

■ Rule-based selection of threads for region execution

```
#pragma omp masked [ filter(integer-expression) ] new-line  
structured-block
```

→ Combined constructs:

→ #pragma omp parallel masked

→ #pragma omp masked taskloop

→ #pragma omp parallel masked taskloop

→ #pragma omp masked taskloop simd

→ #pragma omp parallel masked taskloop simd

Masked construct in OpenMP 5.1 / 2

■ Replacing master construct with more functionality

- OpenMP Construct

```
#pragma omp master  
{}
```

```
#pragma omp masked  
{}
```

```
#pragma omp masked filter(thread_id)  
{}
```

- Semantically equivalent

```
if(omp_get_thread_num()==0)  
{}
```

```
if(omp_get_thread_num()==0)  
{}
```

```
if(omp_get_thread_num()==thread_id)  
{}
```


Masked construct in OpenMP 6.0

- Extend functionality of masked construct to filter for other criterion than thread-id
- Extend usage of filter clause to allow use with other constructs
- Introduce the concept of thread-set, to specify subsets of a current team

```
#pragma omp parallel
{
    // ... define odd to be a thread-set with omp_get_thread_num()%2==1
    #pragma omp for filter(thread-set: odd)
    {}
}
```

Assumes directive

Assumes directive in OpenMP 5.1

- Provides invariants to the implementation to be used for optimizations

- Have to hold at runtime, otherwise behavior is undefined

```
#pragma omp assumes clause [ [ clause ] ... ] new-line
```

- Appear at file or namespace scope

- Clauses:

- absent: guarantee that no such construct will be encountered

- contains: hint that such a construct most likely will be encountered

- holds: scalar-expression evaluates to true in the scope of the directive

- no_openmp: guarantee that no OpenMP related code is executed

Assumes directive in OpenMP 5.1

- Provides invariants to the implementation to be used for optimizations

- Have to hold at runtime, otherwise behavior is undefined

```
#pragma omp assumes clause[ [ [, ] clause] ... ] new-line
```

- Clauses cont'd:

- no_omp_routines: guarantee that no OpenMP API routines will be called

- no_parallelism: guarantee that no OpenMP (implicit or explicit) tasks will be generated and that no SIMD constructs will be executed

- Example:

```
#pragma omp assumes absent(task, taskloop)
```

OpenMP Meta-Programming

The metadirective Directive

- Construct OpenMP directives for different OpenMP contexts
- Limited form of meta-programming for OpenMP directives and clauses

```
#pragma omp target map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
    when( device={arch(nvptx)}: teams loop ) \
    default( parallel loop )
for (i = lb; i < ub; i++)
    v3[i] = v1[i] * v2[i];
```

```
!$omp begin metadirective &
    when( implementation={unified_shared_memory}: target ) &
    default( target map(mapper(vec_map),tofrom: vec) )
!$omp teams distribute simd
do i=1, vec%size()
    call vec(i)%work()
end do
!$omp end teams distribute simd
!$omp end metadirective
```

Nothing Directive

The nothing Directive

- The `nothing` directive makes meta programming a bit clearer and more flexible.
- If a certain criterion matches, the `nothing` directive can stand to indicate that no (other) OpenMP directive should be used.
 - The `nothing` directive is implicitly added if no condition matches

```
!$omp begin metadirective &  
    when( implementation={unified_shared_memory}: &  
          target teams distribute parallel do simd) &  
          default( nothing )  
do i=1, vec%size()  
    call vec(i)%work()  
end do  
!$omp end metadirective
```


Error Directive

Error Directive Syntax

■ Syntax (C/C++)

```
#pragma omp error [clause[[,] clause],...]  
for-loops
```

■ Syntax (Fortran)

```
!$omp error [clause[[,] clause],...]  
do-loops  
[!$omp end loop]
```

■ Clauses

one of: *at*(compilation), *at*(runtime)

one of: *severity*(fatal), *severity*(warning)

message(*msg-string*)

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.

```
#pragma omp parallel
{
    if (omp_get_num_threads() % 2) {
#pragma omp error at(runtime) severity(warning) \
        message("Running on odd number of threads\n");
    }
    do_stuff_that_works_best_with_even_thread_count();
}
```

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.
- More useful in combination with OpenMP metadirective

```
!$omp begin metadirective &  
    when( arch={fancy_processor}: parallel ) &  
    default( error severity(fatal) at(compilation) &  
            message(“No implementation available” )  
    call fancy_impl_for_fancy_processor()  
!$omp end metadirective
```

User Defined Reductions

User Defined Reductions (UDRs)

expand OpenMP's usability

- Use `declare reduction` directive to define operators
- Operators used in reduction clause like predefined ops

```
#pragma omp declare reduction (reduction-identifier : typename-list : combiner) \  
                               [initializer(initializer-expr)]
```

- `reduction-identifier` gives a name to the operator
 - Can be overloaded for different types
 - Can be redefined in inner scopes
- `typename-list` is a list of types to which it applies
- `combiner` expression specifies how to combine values
- `initializer` specifies the operator's identity value
 - `initializer-expression` is an expression or a function

A simple UDR example

■ Declare the reduction operator

```
#pragma omp declare reduction (minindex : index_struct: \  
    (omp_in.value < omp_out.value) ? omp_in : omp_out)  
    initializer(omp_priv = {.value = MAX_INT, .index = 0})
```

■ Use the reduction operator in a reduction clause

```
index_struct min_value = (.value = MAX_INT, .index = 0);  
#pragma omp parallel for reduction (minindex : min_value)  
    for (i = 0; i < NUM_ELEMENTS; i++)  
        if ( a[i] < min_value.value) {  
            min_value.value = a[i]; min_value.index = i;}  
}
```

■ Private copies created for a reduction are initialized to the identity that was specified for the operator and type

→ Default identity defined if `identity` clause not present

■ Compiler uses `combiner` to combine private copies

→ `omp_out` refers to private copy that holds combined value

→ `omp_in` refers to the other private copy

Atomics

OpenMP 5.1 has full featured atomics

- The `atomic` construct accesses a single memory location safely

```
#pragma omp atomic [read|write|update] [capture] [compare] [hint(...)] \  
                  [fail(...)] [seq_cst|acq_rel|release|acquire|relaxed]  
    expression-stmt
```

- `expression-stmt` restricted based on clauses (C/C++ syntax; Fortran similar)
- `atomic-clause` determines types of accesses performed atomically
 - `update` (default): Reads and writes the single memory location atomically
 - `read`: Reads location atomically
 - `write`: Writes location atomically
- Other clauses control details of the accesses and
 - `capture`: Stores value (before or after update) into a private variable
 - `compare`: Specifies that update is performed conditionally (i.e., compare-and-swap semantics)
 - `hint`: Supports optimizations based on expected cross-thread access pattern
 - `fail`: Enables efficient compare-and-swap on some architectures (if compare fails)
 - `memory-order-clause`: Controls extent of atomicity, memory ordering

Clauses on OpenMP atomics support rich feature set of access modalities

- Original OpenMP mechanism could not capture a value atomically

```
int schedule (int upper) {  
    static int iter = 0; int ret;  
    ret = iter;  
    #pragma omp atomic  
        iter++;  
    if (ret <= upper) { return ret; }  
    else { return -1; } //no more iters  
}
```

- Atomic capture provides the needed functionality

```
int schedule (int upper) {  
    static int iter = 0; int ret;  
    #pragma omp atomic update capture  
        ret = iter++; // atomic capture  
    if (ret <= upper) { return ret; }  
    else { return -1; } // no more iters  
}
```

Features support user-level synchronization

■ Naive attempt to write user-level critical section

- Assume `shared_*` are all shared variables
- Assume only two threads access `shared_lock`

```
int local, have_lock = 0;

while (! have_lock) {
    #pragma omp atomic capture
    have_lock = shared_lock++;
}

local =  shared_a;
shared_a = shared_b;
shared_b = local;

#pragma omp atomic write
shared_lock = 0;
```

■ What's wrong with this code?

User-level synchronization must ensure that memory is consistent

■ Correct user-level critical section must include flushes

- Assume `shared_*` are all shared variables
- Assume only two threads access `shared_lock`

```
int local, have_lock = 0;
while (! have_lock) {
    #pragma omp atomic update capture seq_cst
    have_lock = shared_lock++;
}

local =  shared_a;
shared_a = shared_b;
shared_b = local;

#pragma omp atomic write seq_cst
shared_lock = 0;
```

■ Alternatively, must add several flushes (more than 2)

- Could use other memory order clauses but more complicated (and may still need flushes)