OpenMP 5.0 and Beyond:
Taking Good Care of the Node in Exascale?

Dr. Christian Terboven, terboven@itc.rwth-aachen.de
RWTH Aachen University

HUAWEI HPC Workshop 2020

October 29th, 2020

High Performance Computing

OpenMP for Exascale?!

- Why am I talking about OpenMP in the context of Exascale?

- Many believe:
  Exascale programming := MPI + X
  X := OpenMP + Y

- What does it take for OpenMP to be an attractive bride?

- Saying:

  Wear something **old**, something **borrowed**, and something **new**!

Something old

Incremental parallelization and performance optimization

OpenMP

- 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

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
Main topics:

- Affinity
- Tasking
- Tool support
- Accelerator support

Source Example: CG Method

- OpenMP is often used for loop-level parallelism:

```
for (iter = 0; iter < sc->maxIter; iter++) {
    // ...
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    // ...
}
```

- Lets consider the sparse matrix-vector-multiplication:

```
void matvec(Matrix *A, double *x, double *y) {
#pragma omp parallel for private(j,is,ie,j0,y0)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];  ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
}
```

OpenMP allows to …

- … influence the distribution of loop iterations to threads via the `schedule` clause
- ... control thread affinity via the `proc_bind` clause and `OMP_PLACES` environment variable
- ... ensure effective vectorization via the combined loop `simd` construct

- => OpenMP can be simple and complex to apply, depending on your goals

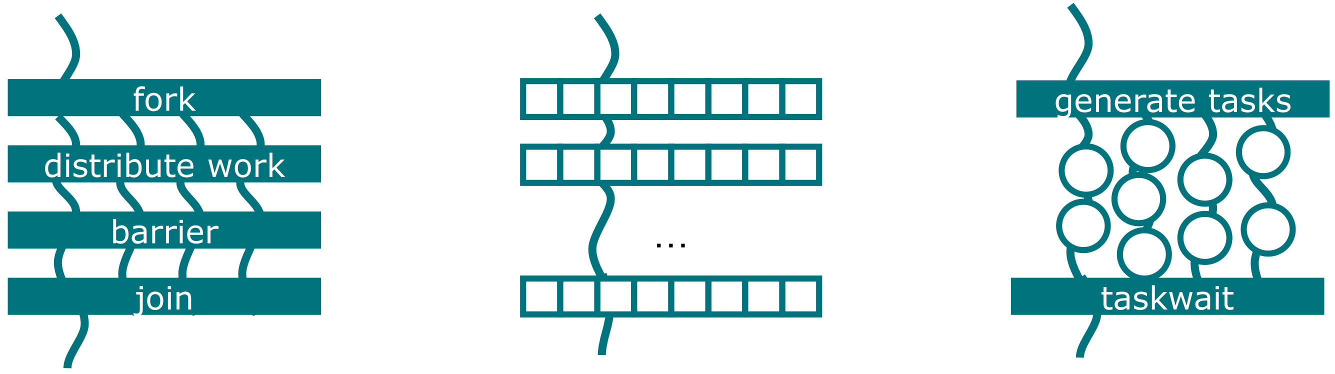
Extending the support for parallel loops

- Existing loop constructs are tightly bound to execution model:

```
#pragma omp parallel for
for (i=0; i<N;++i) {…}
```

```
#pragma omp simd
for (i=0; i<N;++i) {…}
```

```
#pragma omp taskloop
for (i=0; i<N;++i) {…}
```



- OpenMP also allows to …
  - ... implement composable parallelism by means of tasking
  - ... influence the partitioning of loop iteration into tasks via the `grainsize` and `numtasks` clauses
  - … combine all of these models with a well-defined semantic (of interaction)

Source Example: CG Method

- Just one parallel region:

```
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    // ...
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    // ...
}
```

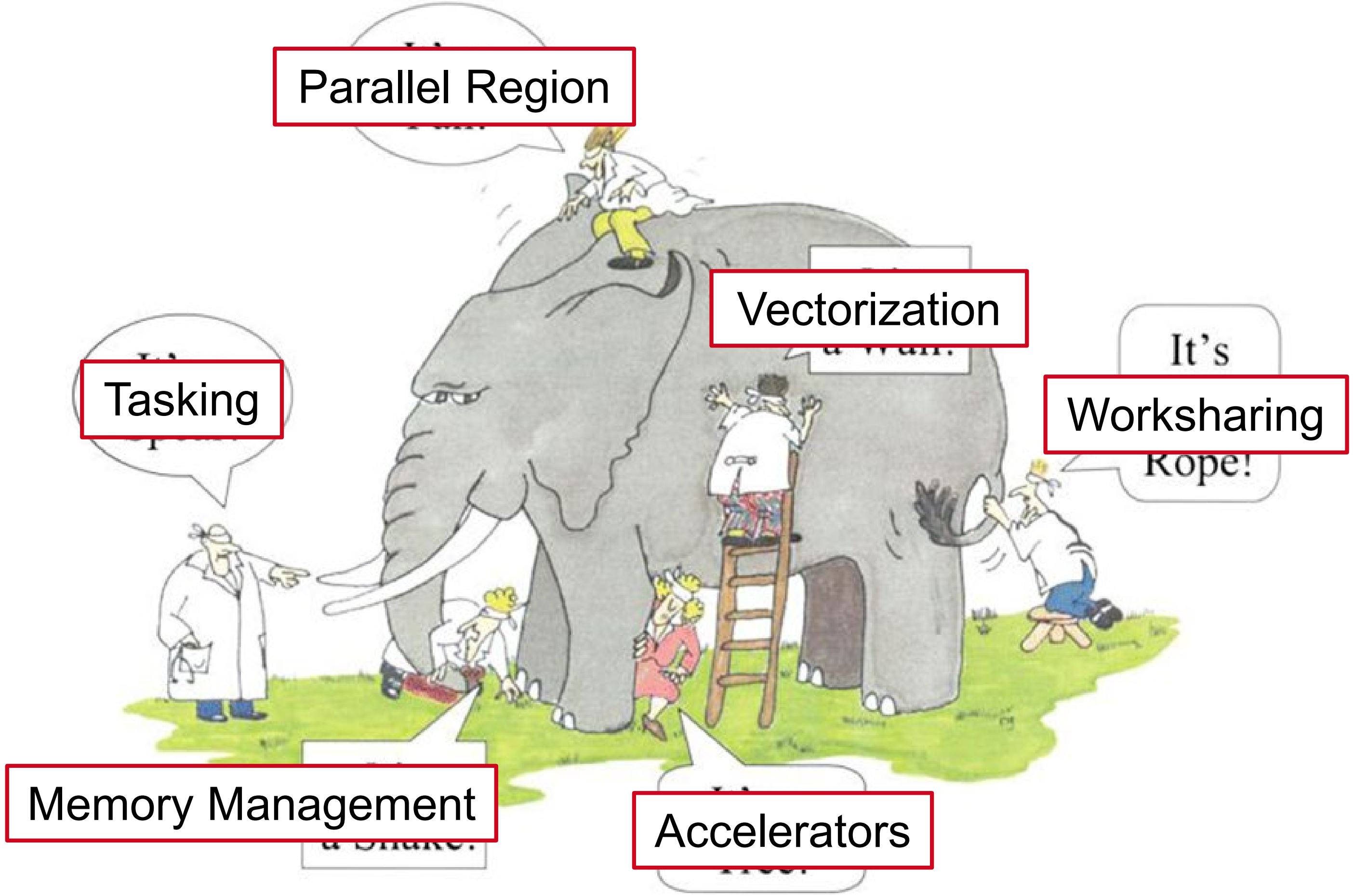
- Re-consider the sparse matrix-vector-multiplication:

```
void matvec(Matrix *A, double *x, double *y) {
#pragma omp taskloop private(j,is,ie,j0,y0)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];  ie = A->ptr[i + 1];
        /* inner for-loop left out for brevity */
        y[i] = y0;
    }
}
```

  – Result: composable parallelism

How did that change OpenMP?

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



Something borrowed / 1

Tasking

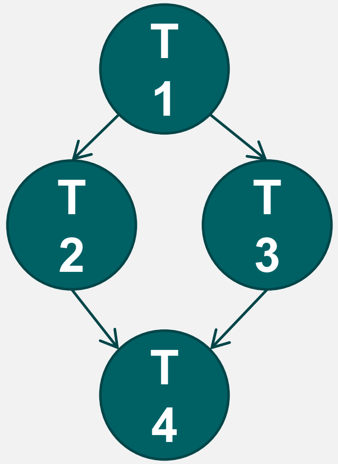
Improvements to Tasking / 1

- A task cannot be executed until all its predecessor tasks are completed

- If a task defin
  – the task will dep … of the list items in an `out` or `inout` d

- If a task defin
  – the task will dep … of the list items in an `in`, `out` or `inc`

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```



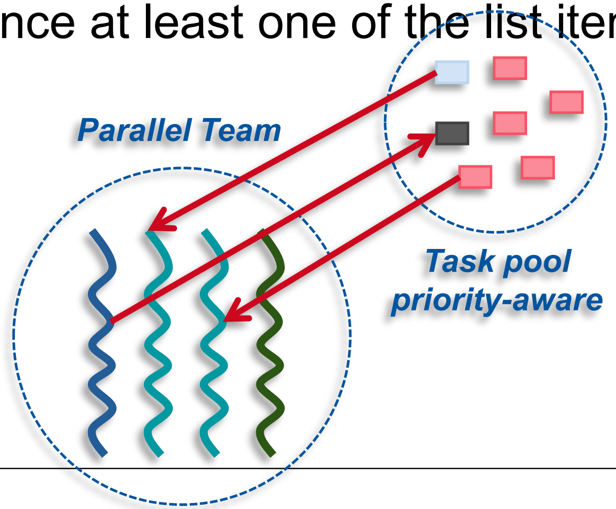
Improvements to Tasking / 1

- A task cannot be executed until all its predecessor tasks are completed

- If a task defines an `in` dependence over a variable
  – the task will depend on all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence

- If a task defines an `out`/`inout` dependence over a variable
  – the task will depend on all previously generated sibling tasks that reference at least one of the list items in an `in`, `out` or `inout` dependence

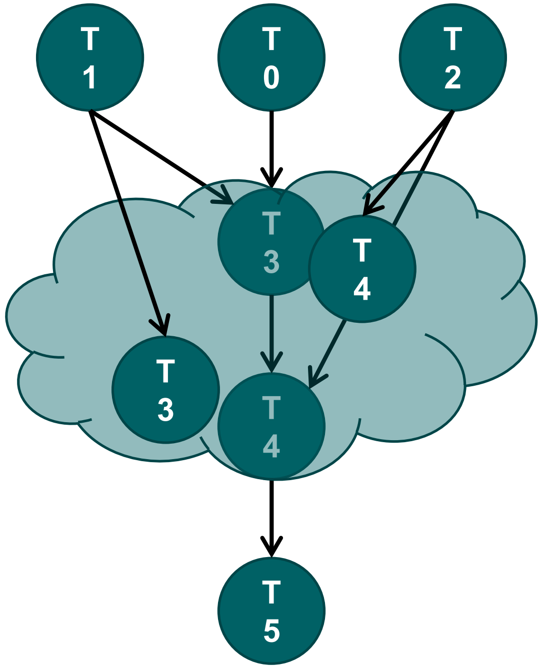
- OpenMP allows to specify Task Priorities to guide execution order



Improvements to Tasking / 2

- New dependency type: `mutexinoutset`

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
   res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res)  //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res)  //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



1. *inoutset property*: tasks with a `mutexinoutset` dependence create a cloud of tasks (an inout set) that synchronizes with previous & posterior tasks that dependent on the same list item

2. *mutex property*: Tasks inside the inout set can be executed in any order but with mutual exclusion

Tasking: the knightly accolade

Jack Dongarra on OpenMP Task Dependencies:

[…] The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. Until now, libraries like PLASMA had to rely on custom built task schedulers; […] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory. **We view OpenMP as the natural path forward for the PLASMA library and expect that others will see the same advantages to choosing this alternative.**

Full article here: http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/

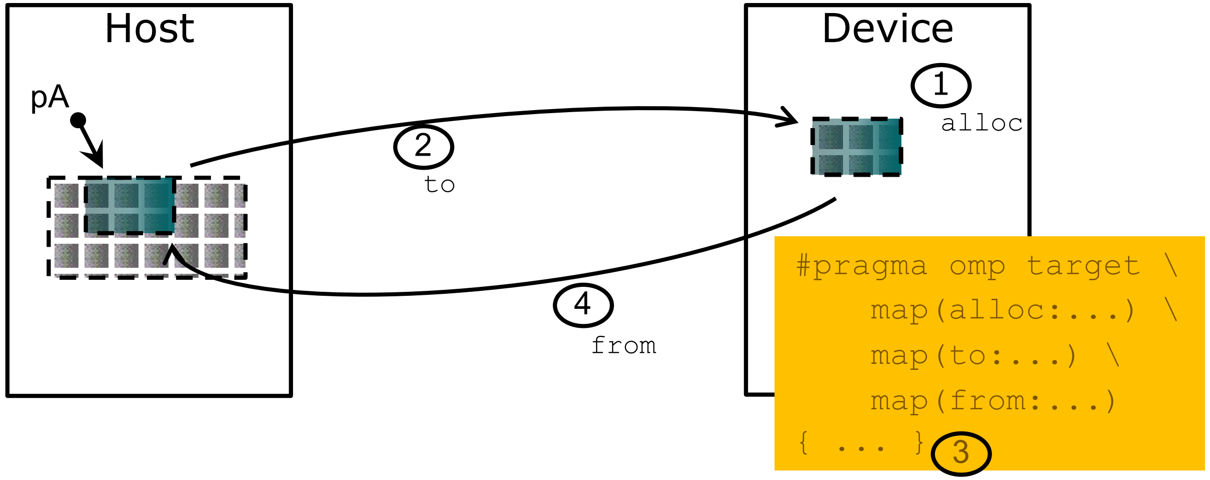
Something borrowed / 2

Accelerators

Offloading

- The `target` construct transfers the control flow to the target device
  - Transfer of control is (by default) sequential and synchronous

- OpenMP separates offload and parallelism



- Support for asynchronicity? Already present in OpenMP: Tasking (see later)

Code snippet: OpenMP vs. OpenACC

- SAXPY: OpenACC compared to OpenMP

OpenACC

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y

  // Run SAXPY
#pragma acc parallel \
 loop gang vector
 for (int i = 0; i < n; i++){
     y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

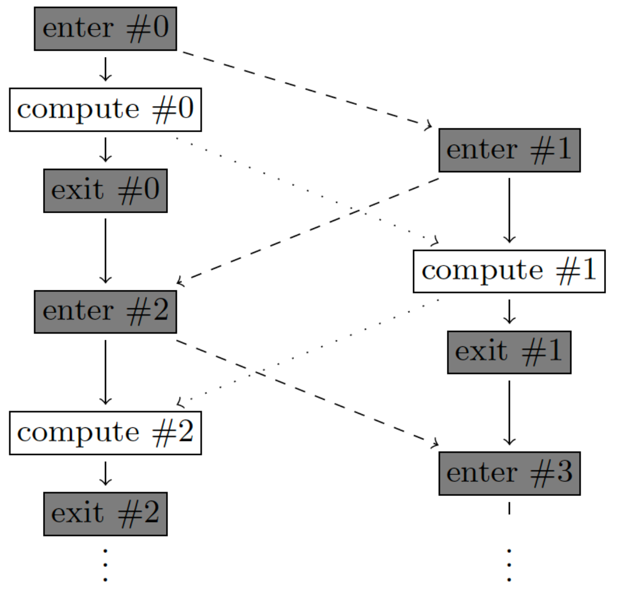
OpenMP

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y

  // Run SAXPY
#pragma omp target
#pragma omp teams distribute parallel for
  for (int i = 0; i < n; i++) {
        y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

Catching up with GPU programming / 1

- OpenMP supports asynchronous and unstructured data movement:

```
double A[ BLOCKS * LEN ];
int enter , compute ;

#pragma omp target enter data nowait map(to: A [0: L
        depend ( out: enter ) depend ( out: A [0: LE
for (int block = 0; block < BLOCKS ; block ++) {
   #pragma omp target enter data nowait depend ( ino
           map (to: A[( block + 1) * LEN: LEN ]) \
           depend (out : A[( block + 1) * LEN: LEN ]
           depend (in: A[( block - 1) * LEN: LEN ])
   #pragma omp target nowait depend ( inout : compute ) \
           map (to: A[ block * LEN: LEN ]) \
           depend ( inout : A[ block * LEN: LEN ])
   {
      // do computation here
   }
   #pragma omp target exit data nowait \
           map ( release : A[ block * LEN: LEN ]) \
           depend ( inout : A[ block * LEN: LEN ])
}
```



Pipelining concept to compute multiple blocks of size len

Strength of OpenMP: integration!

Catching up with GPU programming / 2

- OpenMP 5.1 refines existing functionality
  - Support for mapping (translated) function pointers
  - Device-specific environment variables to control their ICVs
  - The `interop` construct
    - Improves native device support (e.g., CUDA streams)
    - Also supports interoperability with CPU-based libraries (e.g., TBB)
    - Deep-dive OpenMP booth talk by Tom Scogland from LLNL
  - The new `dispatch` construct, improved `declare variant` directive
    - Enable use of variants with device-specific arguments
  - …

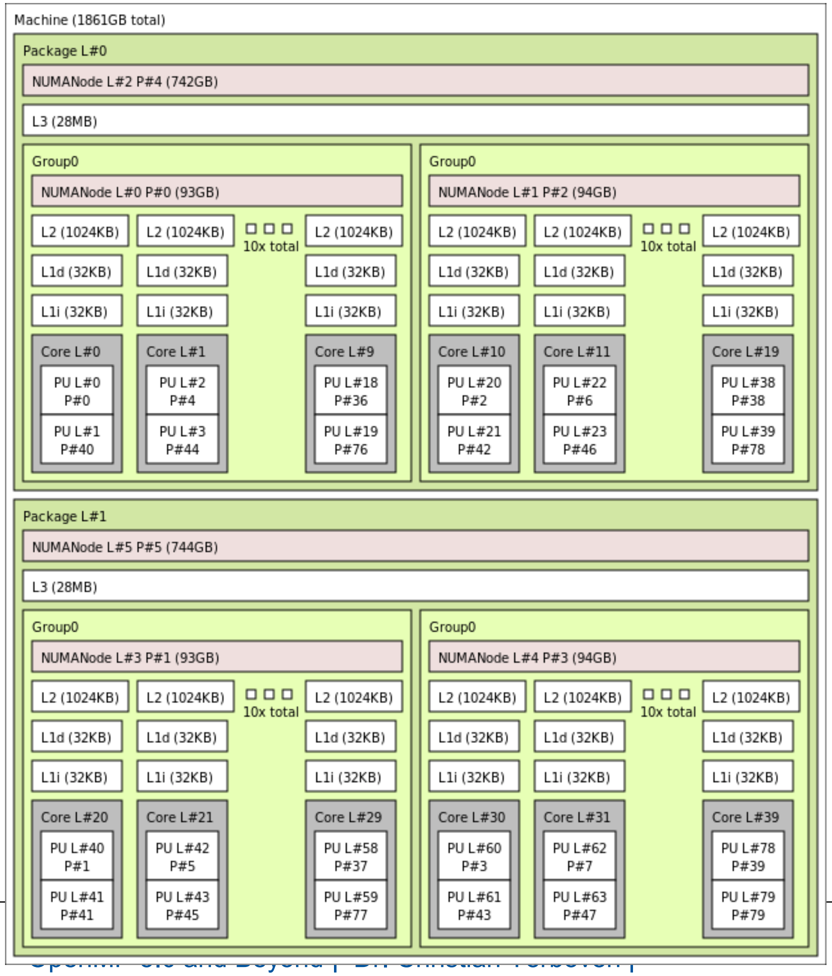
- OpenMP Accelerator subcommittee since 2009:
  - OpenACC's idea: fast GPU-centric development
  - OpenMP' approach: include lessons learnt into OpenMP standard

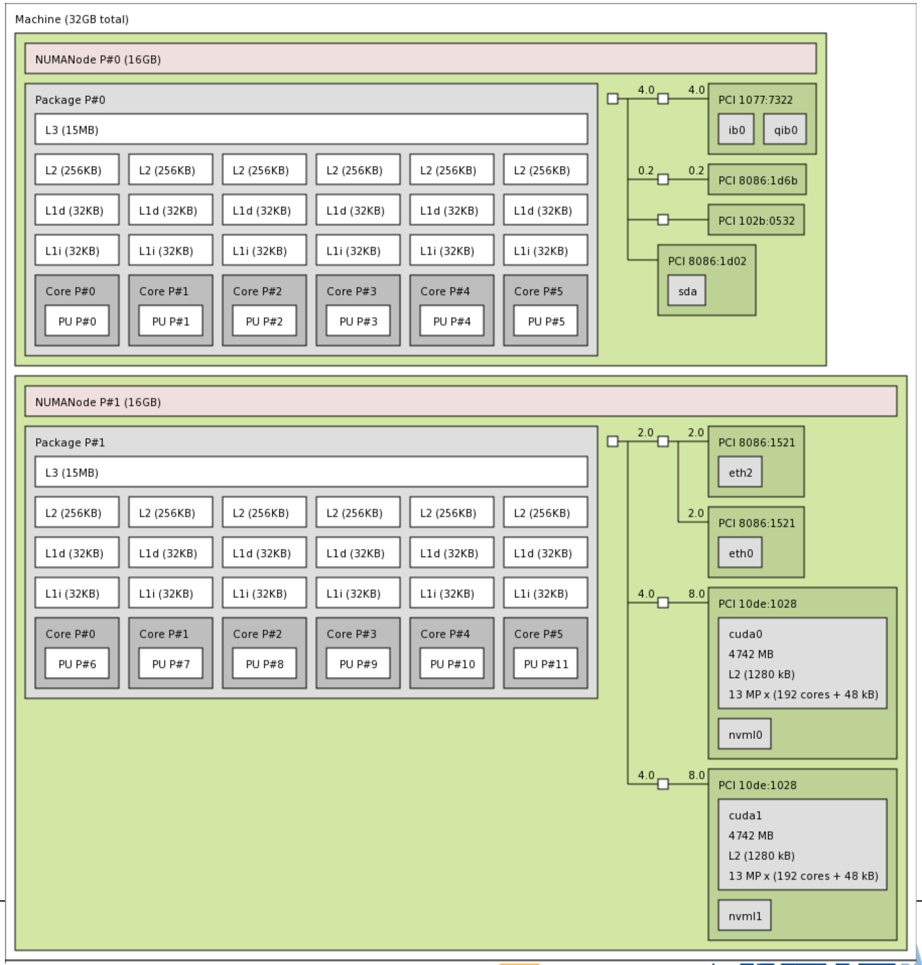
Something new / 1

Memory Management

Memory Management

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





Memory Management: Allocators

- OpenMP 5.0 introduced memory management
  - Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables
    - OpenMP allocators are of type omp_allocator_handle_t
- Allocation:
    - omp_al
- Deallocation
    - omp_fr
    - alloca

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

Something new / 2

Didn't we all miss metaprogramming? ☺

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
```

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
```

The `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

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

Something new / 3

Tool support & *THE* OpenMP Runtime

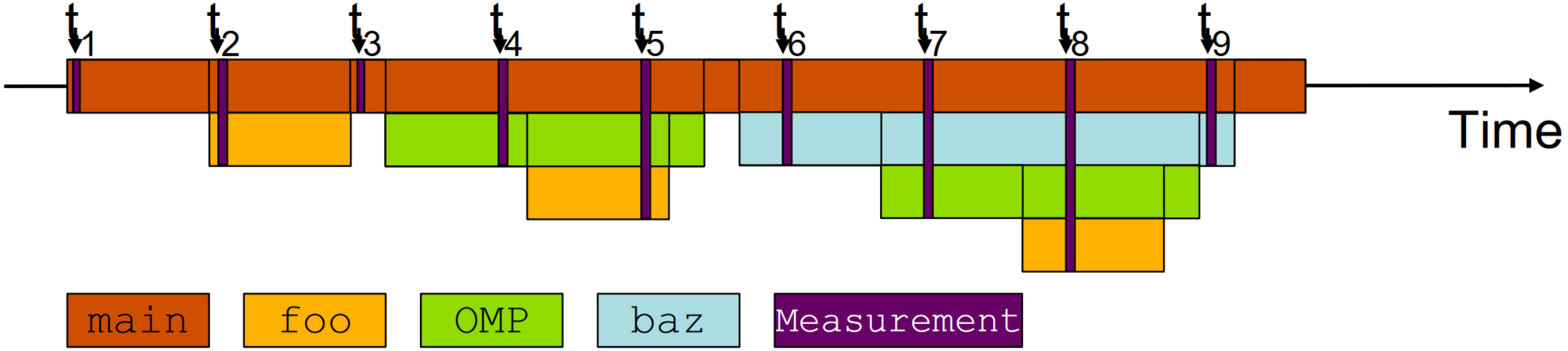
OMPT: Tool support in OpenMP

- For productive performance analysis, performance tools need insight information from the runtime systems!

- OMPT defines states like barrier-wait, work-serial or work-parallel
  - Allows to collect OMPT state statistics in the profile
  - Profile break down for different OMPT states

- OMPT provides frame information
  - Allows to identify OpenMP runtime frames
  - Runtime frames can be eliminated from call trees

```
void foo() {}
void baz() {#omp foo();}
int main()
{foo(); #omp foo();
baz(); return 0;}
```



THE OpenMP Runtime

- In the end of 2012, Intel released their OpenMP runtime as open source
  - https://www.openmprtl.org/
  - Continuous updates since 07/2013 until today
  - Integration into the LLVM project in 08/2013
    - http://openmp.llvm.org/

- This is now *THE* OpenMP runtime for
  - Intel C/C++ and Fortran compilers
  - LLVM compilers
  - GNU compilers (as an alternative)
  - IBM compilers (for OpenPOWER & NVIDIA)
  - numerous research activities and projects
  - …

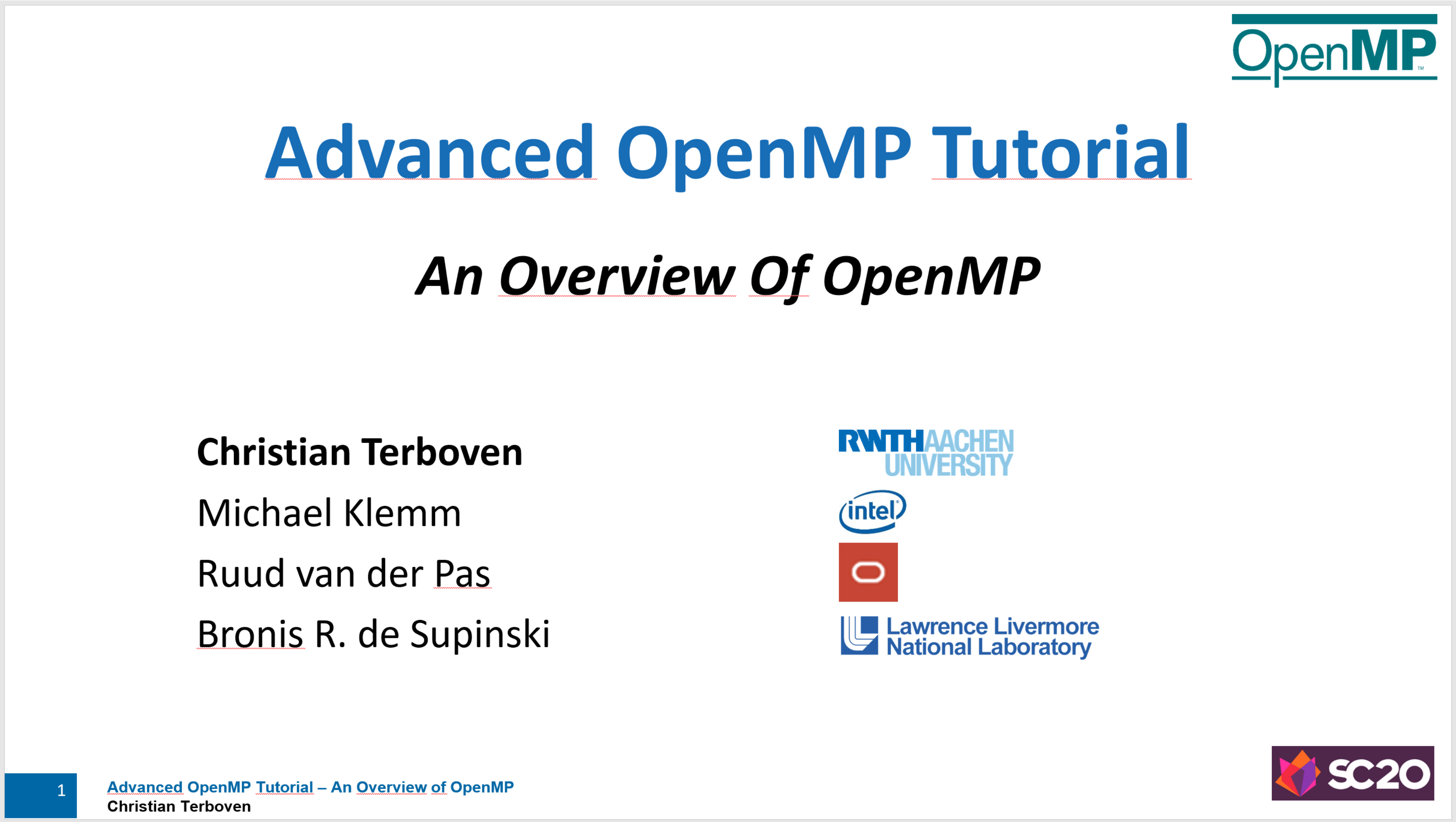
- Available on:
  - Intel & AMD x86
  - IBM OpenPOWER
  - ARM
  - and maybe some I don‘t know of yet

Summary and Outlook

IEEE Proceedings article on vision for OpenMP: "The Ongoing Evolution of OpenMP"

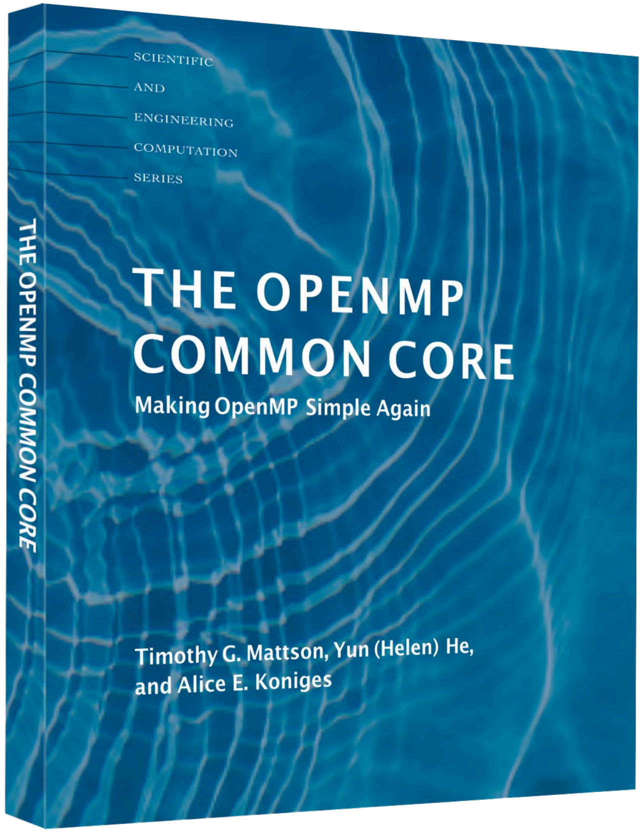
- Broadly support on-node performant, portable parallelism
  - Standardize syntax for commonly available (parallel) directives
  - Consistently apply across C, C++ and Fortran
  - To be simple yet flexible, supporting range of parallelism models
- OpenMP 5.0 fits within that vision
- OpenMP 5.1 refines how OpenMP 5.0 realizes it
- OpenMP 6.0 will be a major step to further realizing it

Advertisement: OpenMP tutorial at SC20



Advertisement: OpenMP books

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



A new book about the OpenMP Common Core, 2019

Projected development

- OpenMP 5.2 will be released by November 2021
  - Late decision during 5.1 process to add this additional minor release
  - Will focus on improving specification of OpenMP syntax
    - Consolidate syntax to highlight commonality and to facilitate use of attributes
    - Clarify and simplify specification of restrictions on clause usage
  - Other changes likely to reduce redundancy in specification

- OpenMP 6.0 will be released in November 2023
  - Deeper support for descriptive and prescriptive control
  - More support for memory affinity and complex hierarchies
  - Support for pipelining, other computation/data associations
  - Continued improvements to device support
    - Extensions of deep copy support (serialize/deserialize functions)
  - Task-only or free-agent threads
  - Event-driven parallelism

OpenMP Support Continues To Increase!

AMD
Greg Rodgers

Argonne National Laboratory
Kalyan Kumaran

ARM
Graham Hunter

ASC/Lawrence Livermore National Laboratory
Bronis R. de Supinski

Barcelona Supercomputing Center
Xavier Martorell

NEC
Shin-ichi Okano

NVIDIA
Jeff Larkin

Oak Ridge National Laboratory
Oscar Hernandez

Red Hat
Torvald Riegel

RWTH Aachen University
Dieter an Mey

Bristol University
Simon McIntosh-Smith

Brookhaven National Laboratory
Vivek Kale

cOMPunity
Yonghong Yan

CRAY
Deepak Eachempati

Edinburgh Parallel Computing Centre
Mark Bull

Sandia National Laboratory
Stephen Olivier

Stony Brook University
Dr. Barbara Chapman

SUSE
Michael Matz

Texas Advanced Computing Center
Kent Milfeld

Texas Instruments
Gaurav Mitra

Fujitsu
Naoki Sueyasu

IBM
Kelvin Li

INRIA
Olivier Aumage

Intel
Xinmin Tian

Lawrence Berkeley National Laboratory
Helen He

The University of Manchester
Antoniu Pop

University of Delaware
Sunita Chandrasekaran

University of Tennessee
Piotr Luszczek

Leibniz Supercomputing Centre
Volker Weinberg

Los Alamos National Laboratory
Jamal Mohd-Yusof

Maui High Performance Computing Center
Alice Koniges

Micron
Randy Meyer

NASA
Henry Jin

OpenMP is widely supported by the industry, as well as the research and academic community

Thank you for your attention.

Questions?