OpenMP API Version 5.0
A Story about Threads, Tasks, and Devices

Michael Klemm
Chief Executive Officer
OpenMP Architecture Review Board
michael.klemm@openmp.org
Disclaimer

- My day time job is being a Principal Engineer at Intel.

- I am an HPC person.

- My view might be (too) skewed towards to the HPC domain.

- This talk might be tainted with my own opinion.
The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.
Membership Structure

- **ARB Member**
  - Highest membership category
  - Participation in technical discussions and organizational decisions
  - Voting rights on organizational topics
  - Voting rights on technical topics (tickets, TRs, specifications)

- **ARB Advisor & ARB Contributors**
  - Contribute to technical discussions
  - Voting rights on technical topics (tickets, TRs, specifications)

Your organization can join and influence the direction of OpenMP. Talk to me or send email to michael.klemm@openmp.org.
OpenMP Roadmap

- OpenMP has a well-defined roadmap:
  - 5-year cadence for major releases
  - One minor release in between
  - (At least) one Technical Report (TR) with feature previews in every year

* Numbers assigned to TRs may change if additional TRs are released.
Levels of Parallelism in the OpenMP API v5.0

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster</td>
<td>Group of computers communicating through fast interconnect</td>
</tr>
<tr>
<td>Coprocessors/Accelerators</td>
<td>Special compute devices attached to the local node through special interconnect</td>
</tr>
<tr>
<td>Node</td>
<td>Group of cache coherent processors communicating through shared memory/cache</td>
</tr>
<tr>
<td>Core</td>
<td>Group of functional units within a die communicating through registers</td>
</tr>
<tr>
<td>Hyper-Threads</td>
<td>Group of thread contexts sharing functional units</td>
</tr>
<tr>
<td>Superscalar</td>
<td>Group of instructions sharing functional units</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Sequence of instructions sharing functional units</td>
</tr>
<tr>
<td>Vector</td>
<td>Single instruction using multiple functional units</td>
</tr>
</tbody>
</table>

Definitions

- The Past: OpenMP < 3.0
- The Present: OpenMP ≥ 3.0 and OpenMP ≤ 5.0
- The Future: OpenMP > 5.0
The Past
(or: Stuff you shouldn’t be doing no more!)
# OpenMP Worksharing

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

    #pragma omp for
    for (i = 0; i < N; i++)
    {
        ...
    }
}
```
double a[N];
double l, s = 0;
#pragma omp parallel for reduction(+:s) \
    private(l) schedule(static, 4)
for (i = 0; i < N; i++)
{
    l = log(a[i]);
    s += l;
}

Good Old Times?

- OpenMP version $\leq 2.5$ standardized the common approach at the time.

- Very simplistic programming that abstracts from the native threading interface.

- Limited scalability due to the effects of Amdahl’s law: serial parts overly limit parallel performance.

- Not suited for the complex algorithms that emerged in the last decade.
The Present
(or: Modern OpenMP)
OpenMP Version 5.0

- OpenMP 5.0 introduced powerful features to improve programmability:
  - Task Reductions
  - Memory Allocators
  - Detachable Tasks
  - Dependence Objects
  - Tools APIs
  - Unified Shared Memory
  - Complete Fortran 2003 Support, Initial Fortran 2008 Support
  - Improved Affinity Support
  - Collapse Non-Rectangular Loops
  - Data Serialization for Offload (Deep Copy)
  - Reverse Offload
  - Improved Task Dependences
  - Task-to-data Affinity
  - Meta-Directives
  - Function Variants
  - Improved Affinity Support
  - Multi-Level Parallelism
  - Task Reductions
  - Loop Construct
  - Parallel Scan
  - Interoperability and Usability Enhancements
The Present
(or: Modern OpenMP)
Task-based Programming
(Modern) Task-based Execution Model

- Supports unstructured parallelism
  - unbounded loops
    ```c
    while ( <expr> ) {
        ...
    }
    ```
  - recursive functions
    ```c
    void myfunc( <args> )
    {
        ...; myfunc( <newargs> ); ...;
    }
    ```

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

Example:
```c
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
}
```
Task Synchronization w/ Dependencies

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    // OpenMP 3.1
    #pragma omp task
    std::cout << x << std::endl;

    #pragma omp task
    long_running_task();

    #pragma omp taskwait

    #pragma omp task
    x++;
}
```

```c
int x = 0;
#pragma omp parallel
#pragma omp single
{
    // OpenMP 4.0
    #pragma omp task depend(in: x)
    std::cout << x << std::endl;

    #pragma omp task
    long_running_task();

    #pragma omp task depend(inout: x)
    x++;
}
```
Example: Cholesky Factorization

```c
void cholesky(int ts, int nt, double* a[nt][nt]) {
    for (int k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf(a[k][k], ts, ts);

        // Triangular systems
        for (int i = k + 1; i < nt; i++) {
            #pragma omp task
            trsm(a[k][k], a[k][i], ts, ts);
        }

        #pragma omp taskwait

        // Update trailing matrix
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
                #pragma omp task
                dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
            }
            #pragma omp task
            syrk(a[k][i], a[i][i], ts, ts);
        }
    }
}
```

OpenMP 3.1

OpenMP 4.0
Example: saxpy Operation

- Manual transformation is cumbersome and error prone
- Applying blocking techniques for large loops can be tricky
- taskloop: improved programmability

```c
for (i = 0; i < SIZE; i += 1) {
}
```

```c
#pragma omp taskloop grainsize(TS)
for (i = 0; i < SIZE; i += 1) {
}
```
Example: Sparse CG w/ taskloop

```c
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance) break;
    rho_old = rho;
}

void matvec(Matrix *A, double *x, double *y) {
    // ...
#pragma omp taskloop private(j, is, ie, j0, y0) grain_size(grainsz)
    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;
    }
    // ...
}
```
Task Reductions

- Task reductions extend traditional reductions to arbitrary task graphs
- Extend the existing task and taskgroup constructs
- Also work with the taskloop construct

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        {
            while (node) {
                #pragma omp task in_reduction(+: res) \ firstprivate(node)
                {
                    res += node->value;
                }
                node = node->next;
            }
        }
    }
}
```
The Present
(or: Modern OpenMP)
Heterogeneous Programming for Coprocessors
Device Model

- OpenMP 4.0 supports accelerators/coprocessors, aka heterogeneous programming

- Device model:
  - One host
  - Multiple accelerators/coprocessors of the same kind
Execution Model

- The target construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - The transfer clauses control direction of data flow
  - Array notation is used to describe array length

- The target data construct creates a scoped device data environment
  - Does not include a transfer of control
  - The transfer clauses control direction of data flow
  - The device data environment is valid through the lifetime of the target data region

- Use target update to request data transfers from within a target data region
Example

```c
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
    #pragma omp target device(0)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

    #pragma omp target update device(0) to(input[:N])

    #pragma omp target device(0)
    #pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i)
}
```
Multi-level Device Parallelism

```c
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, b & initialize x, y

    #pragma omp target data map(to:x[0:n])
    {
        #pragma omp target map(tofrom:y)
        #pragma omp teams num_teams(num_blocks) num_threads(bsize)
        all do the same
        #pragma omp distribute
        for (int i = 0; i < n; i += num_blocks){
            workshare (w/o barrier)
        }
        #pragma omp parallel for
        for (int j = i; j < i + num_blocks; j++) {
            workshare (w/ barrier)
            y[j] = a*x[j] + y[j];
        }
    }
}
```
```c
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, b & initialize x, y

    #pragma omp target map(to:x[0:n]) map(tofrom:y)
    {
        #pragma omp teams distribute parallel for \
            num_teams(num_blocks) num_threads(bsize)
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
}
```
Loop Construct

- Existing loop constructs are tightly bound to execution model:
  - `#pragma omp 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) {...}

- The loop construct is meant to let the OpenMP implementation pick choose the right parallelization scheme.
Simplifying Multi-level Device Parallelism

```c
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, b & initialize x, y

    #pragma omp target map(to:x[0:n]) map(tofrom:y)
    {
        #pragma omp loop
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
}
```
The Present
(or: Modern OpenMP)
Controlling the Memory Hierarchy
Memory Allocators

- New clause on all constructs with data sharing clauses:
  - `allocate([allocator:] list )`

- Allocation:
  - `omp_alloc(size_t size, omp_allocator_t *allocator)`

- Deallocation:
  - `omp_free(void *ptr, const omp_allocator_t *allocator)`
  - `allocator` argument is optional

- `allocate` directive
  - Standalone directive for allocation, or declaration of allocation stmt.
Example: Using Memory Allocators

```c
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR and/or omp_set_default_allocator
    double *p = (double *) malloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem Alloc:c) // on target; must be compile-time expr
    {
        #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
        {
            some_other_parallel_code();
        }
    }

    omp_free(p);
}
```
The Future
(or: Post-modern OpenMP)
Continuum of Control

- **Descriptive**
  - Express “what”
  - Ignore implementation
  - Rely on quality of implementation

- **Prescriptive**
  - Express “how”
  - Focus on implementation
  - Expose control over execution

**OpenMP** strives to
- Support a useful subset of this spectrum
- Provide a structured path from descriptive to prescriptive where needed

```
#pragma omp task
#pragma omp loop
#pragma omp task
#pragma omp for
#pragma omp for
#pragma omp for

schedule(static,5)
```
OpenMP API Version 5.1

- OpenMP 5.0 evolved the OpenMP API quite considerably
- Version 5.1 will refine OpenMP 5.0 features
- Plus: clarifications, corrections, editing, etc.
- No big additions; vendors need time for high-quality implementations
OpenMP API Version 5.1

- Improved C++ support through attribute syntax

- Utility directives, e.g., `error`
  - Print diagnostic information at compile time or runtime
  - May include severity clause: `fatal` or `warning`

- Improved native device support (e.g., CUDA streams)

- Language-level subset of OpenMP (inverse of `requires`)
OpenMP API Version 6.0

- Support for descriptive specification and prescriptive control

- Improvements for memory affinity and complex memory hierarchies/traits

- Free-agent threads, relaxing the notion of thread teams

- Event-driven parallelism

- Completed support for new normative references
Adverts: Engage with the OpenMP Community
OpenMPCon & IWOMP 2019

**Dates:**
- OpenMPCon: Sep 9 – 10
- Tutorials: Sep 11
- IWOMP: Sep 12-13

**Location:**
- University of Auckland

**General Chair:**
- Dr. Oliver Sinnen
- PARC lab
- Department of Electrical and Computer Engineering
- University of Auckland
Tutorials at Supercomputing 2019

- OpenMP Common Core: A “Hands-On” Exploration
  - Barbara Chapman, Helen He, Alice Koniges, Tim Mattson,

- Mastering Tasking with OpenMP
  - Michael Klemm, Christian Terboven, Xavier Teruel, Bronis de Supinski

- Advanced OpenMP: Performance and 5.0 Features
  - Michael Klemm, Christian Terboven, Bronis de Supinski, Ruud van der Pas

- Programming Your GPU with OpenMP: A Hands-On Introduction
  - Simon McIntosh-Smith, Tim Mattson
The Last Slide

- OpenMP 5.0 was a major leap forward
  - Maybe the biggest release ever in the history of OpenMP
  - Well-defined interfaces for tools
  - New ways to express parallelism, improved usage of existing features

- OpenMP is a modern directive-based programming model
  - Multi-level parallelism supported (coprocessors, threads, SIMD)
  - Task-based programming model is the modern approach to parallelism
  - Powerful language features for complex algorithms
  - High-level access to parallelism; path forward to highly efficient programming