Past, Present, and Future of OpenMP*
(An OpenMP Carol)

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group
(michael.klemm@intel.com)

*Other brands and names are the property of their respective owners.
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015 Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, and Cilk are trademarks of Intel Corporation in the U.S. and other countries. *Other names and brands may be claimed as the property of others.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
</tbody>
</table>

Notice revision #20110804
Very Short Agenda

• The Past  OpenMP ≤ 3.0
• The Present  OpenMP 4.0
• The Future  OpenMP 4.1
The Past
OpenMP API

- De-facto standard, OpenMP 4.0 out since July 2013
- API for C/C++ and Fortran for shared-memory parallel programming
- Based on directives (pragmas in C/C++)
- Portable across vendors and platforms
- Supports various types of parallelism
OpenMP History

In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for fortran is released.

OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP's cooperative style of operation.

OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.

OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.

1996: Vendors provide similar but different solutions for loop parallelism, causing portability and maintenance problems. Kuck and Associates (KAI) | SGI | Cray | IBM | High Performance Fortran (HPF) | Parallel Computing Forum (PCF)

- **1997**: 55 pages
- **1998**: 76 pages
- **1999**: 116 pages

**1997**
- **Loop Parallelization**: 1.0
  - First hybrid applications with MPI and OpenMP appear.
- **C++**
  - 1.0
  - Merger of Fortran and C/C++ specifications begins.

**1998**
- **Tasking**: 2.5
  - Unified C/C++ and Fortran: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.

**1999**
- **Heterogeneity**: 3.1
  - Supports min/max, reductions in C/C++

**2000**
- **Unified**: 3.0
  - Incorporates task parallelism—a hard problem as OpenMP struggles to maintain its thread-based nature while accommodating the dynamic nature of tasking.

**2001**
- **Supports**: 4.0
  - Supports accelerator/coprocessor devices, SIMD parallelism, thread affinity, and more. Expands OpenMP beyond its traditional boundaries.

**OpenMP ARB Membership Evolution**
- **Permanent ARB**: Blue
- **Auxiliary ARB Members**: Light Blue
- **OpenMP Google Scholar Hits**: Red

**SOFTWARE AND SERVICES**
The Past in Three Slides (1)

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

SOFTWARE AND SERVICES
The Past in Three Slides (2)

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;
}

SOFTWARE AND SERVICES
The Past in Three Slides (3)

#pragma omp parallel
#pragma omp single
for(e = l->first; e; e = e->next)
#pragma omp task
process(e);

fork

join
The Present
## OpenMP of the Present

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Group of computers communicating through fast interconnect</th>
</tr>
</thead>
<tbody>
<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 processors communicating through shared memory</td>
</tr>
<tr>
<td>Socket</td>
<td>Group of cores communicating through shared cache</td>
</tr>
<tr>
<td>Core</td>
<td>Group of functional units 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>

**OpenMP 4.0 for Devices**

| OpenMP 4.0 SIMD |

**SOFTWARE AND SERVICES**
OpenMP 4.0 SIMD
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
In a Time before OpenMP 4.0

• Programmers had to rely on auto-vectorization...
• ... or to use vendor-specific extensions
  • Programming models (e.g., Intel® Cilk™ Plus)
  • Compiler pragmas (e.g., #pragma vector)
  • Low-level constructs (e.g., _mm_add_pd())

```c
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

You need to trust the compiler to do the “right” thing.
OpenMP 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 [for] simd [clause[[], clause],…] for-loops

• Syntax (Fortran)
  !$omp [do] simd [clause[[], clause],…] do-loops
Example

```c
void 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;
}
```

parallelize

vectorize

Thread 0

Thread 1

Thread 2

SOFTWARE AND SERVICES
Data Sharing Clauses

• `private(var-list)`: Uninitialized vectors for variables in `var-list`
  
  x: 42 → ? ? ? ?

• `firstprivate(var-list)`: Initialized vectors for variables in `var-list`
  
  x: 42 → 42 42 42 42

• `reduction(op:var-list)`: Create private variables for `var-list` and apply reduction operator `op` at the end of the construct
  
  12 5 8 17 → x: 42
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
    \[ x_i = x_{\text{orig}} + i \times \text{linear-step} \]
- **aligned** (*list[:alignment]*)
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- **collapse** (*n*)
SIMD Function Vectorization

```c
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

```c
#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]);
    }
}

vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}

vec8 distsq_v(vec8 x, vec8 y) {
    return (x - y) * (x - y);
}

vd = min_v(distsq_v(va, vb, vc))
```
SIMD Function Vectorization

- **simdlen** \((\text{length})\)
  - generate function to support a given vector length
- **uniform** \((\text{argument-list})\)
  - 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** \((\text{argument-list}[::\text{linear-step}])\)
- **aligned** \((\text{argument-list}[::\text{alignment}])\)
- **reduction** \((\text{operator}:\text{list})\)
SIMD Constructs & Performance


SOFTWARE AND SERVICES
OpenMP 4.0 for Devices
Device Model

• OpenMP 4.0 supports accelerators/coprocessors
• Device model:
  • One host
  • Multiple accelerators/coprocessors of the same kind
OpenMP 4.0 for Devices - Constructs

- Transfer control [and data] from the host to the device

- Syntax (C/C++)
  
  #pragma omp target [data] [clause[, clause],…] structured-block

- Syntax (Fortran)
  
  !$omp target [data] [clause[, clause],…] structured-block
  !$omp end target [data]

- Clauses
  
  device(scalar-integer-expression)
  map([alloc | to | from | tofrom:] list)
  if(scalar-expr)
Execution Model

• The \textbf{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 \textbf{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 \textbf{target update} to request data transfers from within a target data region
Execution Model

- Data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released

```c
#pragma omp target 
   map alloc: (...) \ 
   map to: (...) \ 
   map from: (...) 
{ ... } 
```
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)
}
```
teams Construct

• Support multi-level parallel devices

• Syntax (C/C++):
  
  ```c
  #pragma omp teams [clause[[,] clause],...]
  structured-block
  ```

• Syntax (Fortran):
  
  ```fortran
  !$omp teams [clause[[,] clause],...]
  structured-block
  ```

• Clauses
  
  ```
  num_teams(integer-expression)
  num_threads(integer-expression)
  default(shared | none)
  private(list), firstprivate(list)
  shared(list), reduction(operator : list)
  ```
Offloading SAXPY to a Coprocessor

```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(nthreads)

        for (int i = 0; i < n; i += num_blocks){
            for (int j = i; j < i + num_blocks; j++) {
                y[j] = a*x[j] + y[j];
            }
        }
    }
    free(x); free(y); return 0;
}
```

all do the same
Offloading SAXPY to a Coprocessor

```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)

        #pragma omp distribute
        for (int i = 0; i < n; i += num_blocks){
            #pragma omp parallel for
            for (int j = i; j < i + num_blocks; j++) {
                y[j] = a*x[j] + y[j];
            }
        }
    }
    free(x); free(y); return 0; }
```
Offloading SAXPY to a Coprocessor

```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];
        }
    }

    free(x); free(y); return 0;
}
```
The Future
Task-generating Loops
Issues with Traditional Worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel \texttt{dgemm} in MKL

```c
void example() {
    #pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // \texttt{dgemm} is either parallel or sequential
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);
    }
}
```

- Writing such code either
  - oversubscribes the system,
  - yields bad performance due to OpenMP overheads, or
  - needs a lot of glue code to use sequential \texttt{dgemm} only for sub-matrixes
The **taskloop** Construct

- Parallelize a loop using OpenMP tasks
  - Cut loop into chunks
  - Create a task for each loop chunk

- **Syntax (C/C++)**
  ```
  #pragma omp taskloop [simd] [clause[[,] clause],...]
  for-loops
  ```

- **Syntax (Fortran)**
  ```
  !$omp taskloop[simd] [clause[[,] clause],...]
  do-loops
  !$omp end taskloop [simd]
  ```
Clauses for **taskloop** Construct

- Taskloop constructs inherit clause both from worksharing constructs and the **task** construct
  - shared, private
  - firstprivate, lastprivate
  - default
  - collapse
  - final, untied, mergeable

- **grainsize**(grain-size)
  Chunks have at least grain-size and max 2*grain-size loop iterations

- **num_tasks**(num-tasks)
  Create num-tasks tasks for iterations of the loop
Example: Sparse CG

```c
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;
}
```

```c
void matvec(Matrix *A, double *x, double *y) {
    // ...
    #pragma omp parallel for \
    private(i,j,is,ie,j0,y0) \
    schedule(static)
    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;
    }
    // ...
}
```
Example: Sparse CG

```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;
}
```

```c
void matvec(Matrix *A, double *x, double *y) {
    // ...
    return;
}
```

```c
#pragma omp taskloop private(j,is,ie,j0,y0) grain_size(500)
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;
}
// ...
```
Performance of Sparse CG w/ Tasks

(c) MareNostrum III

(d) Gothmog

X. Teruel, M. Klemm, K. Li, X. Martorell, S.L. Olivier, and C. Terboven. A Proposal for Task-Generating Loops in OpenMP. In A.P. Rendell et al., editor, International Workshop on OpenMP, pages 1-14, Canberra, Australia, September 2013. LNCS 8122
Locks with Hints
Motivation

• Hardware supports new concepts for locks
  • Intel® Transactional Synchronization Extensions
  • Transactional memory in BlueGene*/Q

• Coarse-grained control does not help applications that have mixed locking requirements
  • Some locks may be highly contended
  • Some locks may be used to protect system calls (e.g., IO)
  • Some locks may be just there for safety, but are almost never conflicting (e.g., hash map)

• Programmers need the ability to choose locks on a per-use basis
Two new API Routines

- `omp_init_lock(omp_lock_t *lock)`
- `omp_init_lock_with_hint(omp_lock_t *lock, omp_lock_hint_t hint)`
- `omp_set_lock(omp_lock_t *lock)`
- `omp_unset_lock(omp_lock_t *lock)`
- `omp_destroy_lock(omp_lock_t *lock)`
Two new API Routines

- `omp_init_nest_lock(omp_nest_lock_t *lock)`
- `omp_init_nest_lock_with_hint(omp_nest_lock_t *lock, omp_lock_hint_t hint)`
- `omp_set_nest_lock(omp_nest_lock_t *lock)`
- `omp_unset_nest_lock(omp_nest_lock_t *lock)`
- `omp_destroy_nest_lock(omp_nest_lock_t *lock)`
Hints

• Hints are integer expressions
  • C/C++: can be combined using the | operator
  • Fortran: can be combined using the + operator

• Supported hints:
  • omp_lock_hint_none
  • omp_lock_hint_uncontended
  • omp_lock_hint_contended
  • omp_lock_hint_nonspeculative
  • omp_lock_hint_speculative
New Clause for critical Construct

• Specify a hint how to implement mutual exclusion
  • If a hint clause is specified, the critical construct must be a named construct.
  • All critical constructs with the same name must have the same hint clause.
  • The expression of the hint clause must be a compile-time constant.

• Syntax (C/C++)
  #pragma omp critical [(name)] [hint(expression)] structured-block

• Syntax (Fortran)
  !$omp critical [(name)] [hint(expression)] structured-block
  !$omp end critical [(name)]
void example_locks() {
    omp_lock_t lock;
    omp_init_lock_with_hint(&lock, omp_hint_speculative);
    #pragma omp parallel
    {
        omp_set_lock(&lock);
        do_something_protected();
        omp_unset_lock(&lock);
    }
}

void example_critical() {
    #pragma omp parallel for
    for (int i = 0; i < upper; ++i) {
        Data d = get_some_data(i);
        #pragma omp critical (HASH) hint(omp_hint_speculative)
        hash.insert(d);
    }
}
Using Hints May Increase Performance

- Blindly using speculative locks does not help (KMP_LOCK_KIND=...)
- Speculative locks can benefit more with growing thread counts

Device Constructs
Asynchronous Offloading in 4.0

• You can this at your own risk 😊

```c
#pragma omp parallel sections num_threads(2)
{
    #pragma omp task
    {
        #pragma omp target map(to:input[:N]) map(from:result[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++) {
            result[i] = some_computation(input[i], i);
        }
    }
    #pragma omp task
    {
        do_something_important_on_host();
    }
    #pragma omp taskwait
}
```
Asynchronous Offloading in 4.1

- OpenMP 4.1 requires much less coding and has much cleaner semantics

```c
#pragma omp target map(to:input[:N]) map(from:result[:N]) nowait
#pragma omp parallel for
    for (i=0; i<N; i++) {
        result[i] = some_computation(input[i], i);
    }

do_something_important_on_host();
```
OpenMP 4.1 for Devices

- Transfer control [and data] from the host to the device

- Syntax (C/C++)
  #pragma omp target [data] [clause[[,] clause],…] structured-block

- Syntax (Fortran)
  !$omp target [data] [clause[[,] clause],…] structured-block
  !$omp end target [data]

- General clauses (since OpenMP 4.0)
  device(scalar-integer-expression)
  map([alloc | to | from | tofrom:] list)
  if(scalar-expr)

- Clauses for asynchronous offloading (also supported by target update)
  nowait
  depend(dependency-type:list)

SOFTWARE AND SERVICES
Creating and Destroying Device Data

```c
struct DeviceBuffer {
    // ...
    DeviceBuffer(int dev, size_t sz) {
        #pragma omp target enter data device(dev) map(alloc:buffer[:sz])
    }
    ~DeviceBuffer() {
        #pragma omp target exit data device(dev) map(delete:buffer[:sz])
    }
}

void example() {
    DeviceBuffer *buf1 = new DeviceBuffer(0, 1024);
    compute_a_lot_using_offloading(buf1);
    DeviceBuffer *buf2 = new DeviceBuffer(0, 2048);
    compute_some_more_using_offloading(buf1, buf2);
    delete buf1;
    compute_evenmore_using_offloading(buf2);
    delete buf2;
}
```
Creating and Destroying Device Data

• Manage data without being bound to scoping rules

• Syntax (C/C++)
  #pragma omp target enter data [clause[|,] clause],...]
  #pragma omp target exit data [clause[|,] clause],...]

• Syntax (Fortran)
  !$omp target enter data [clause[|,] clause],...]
  !$omp target exit data [clause[|,] clause],...]

• Clauses
  device(scalar-integer-expression)
  map([alloc | delete | to | from | tofrom:] list)
  if(scalar-expr)
  depend(dependency-type:list)
  nowait
We’re Almost Through

• There are so many things in OpenMP today
  • Can’t cover all of them in an hour!

• OpenMP 4.0 and 4.1 have more to offer!
  • Improved Fortran 2003 support
  • Affinity
  • User-defined reductions
  • Task dependencies
  • Cancellation

• We can chat about these features during the coffee breaks (if you want to)
The last Slide...

- OpenMP 4.0 was a major leap for OpenMP
  - New kind of parallelism has been introduced
  - Support for heterogeneous systems with coprocessor devices

- OpenMP 4.1 will not only be a bugfix release
  - Task-generating loops
  - Locks with hints
  - Feature freeze completed; comment draft almost ready

- OpenMP 5.0 has already been started
  - Expected release during Supercomputing 2017