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

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

Legal Disclaimer & Optimization Notice

Optimization Notice
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. Notice revision #20110804

Very Short Agenda

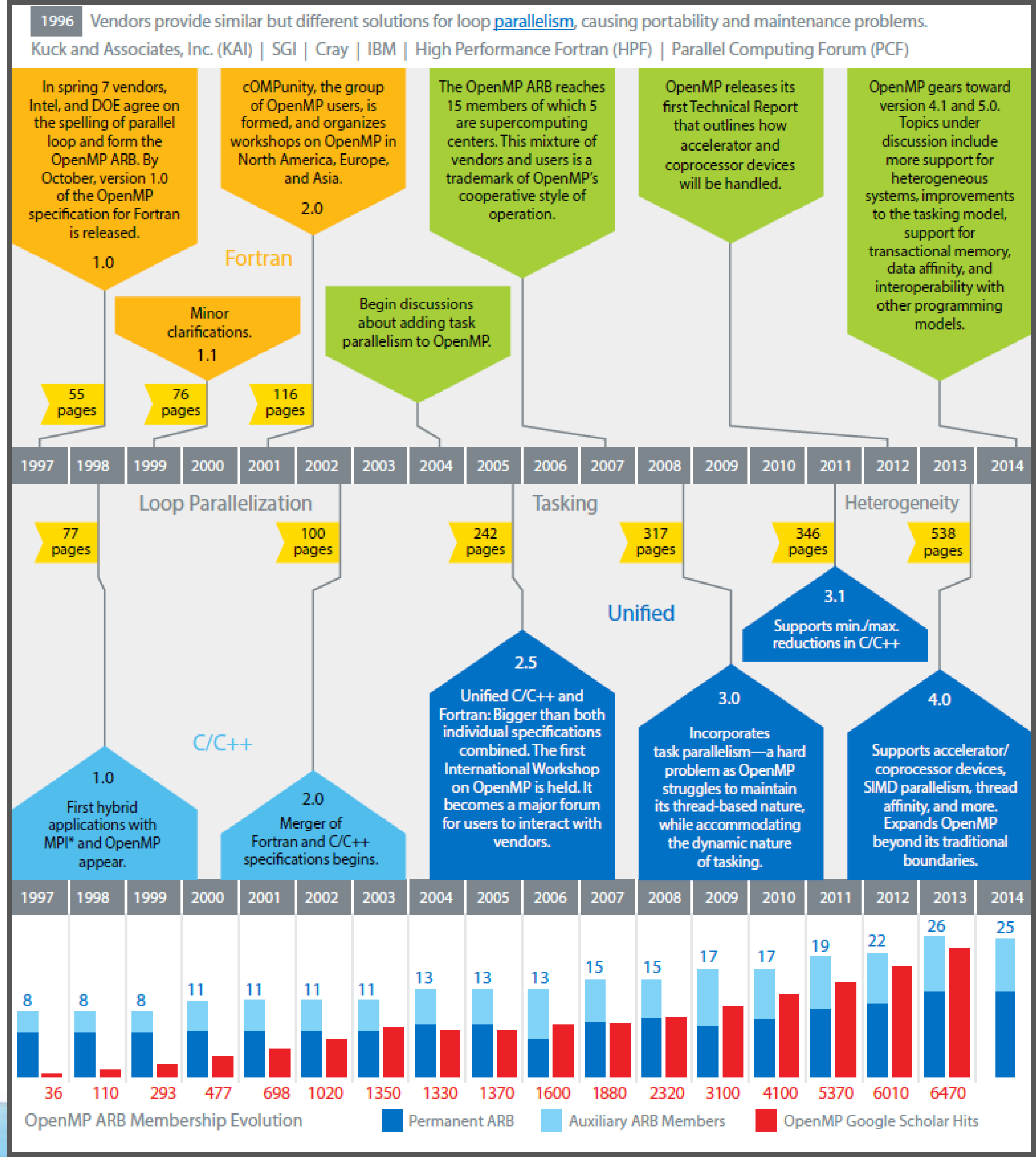
- The Past OpenMP $\leq$ 3.0
- The Present OpenMP 4.0
- The Future OpenMP 4.1

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

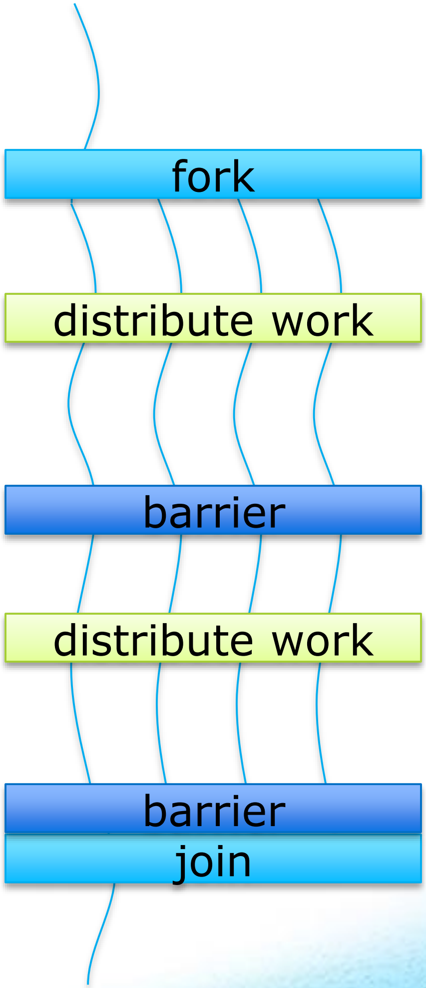


SOFTWARE AND SERVICES

The Past in Three Slides (1)

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

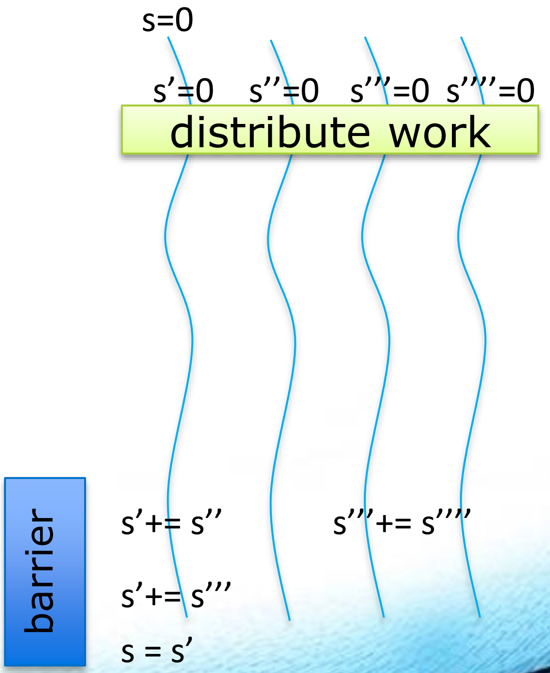
   #pragma omp for
   for (i = 0; i< N; i++)
   {…}
}
```



The Past in Three Slides (2)

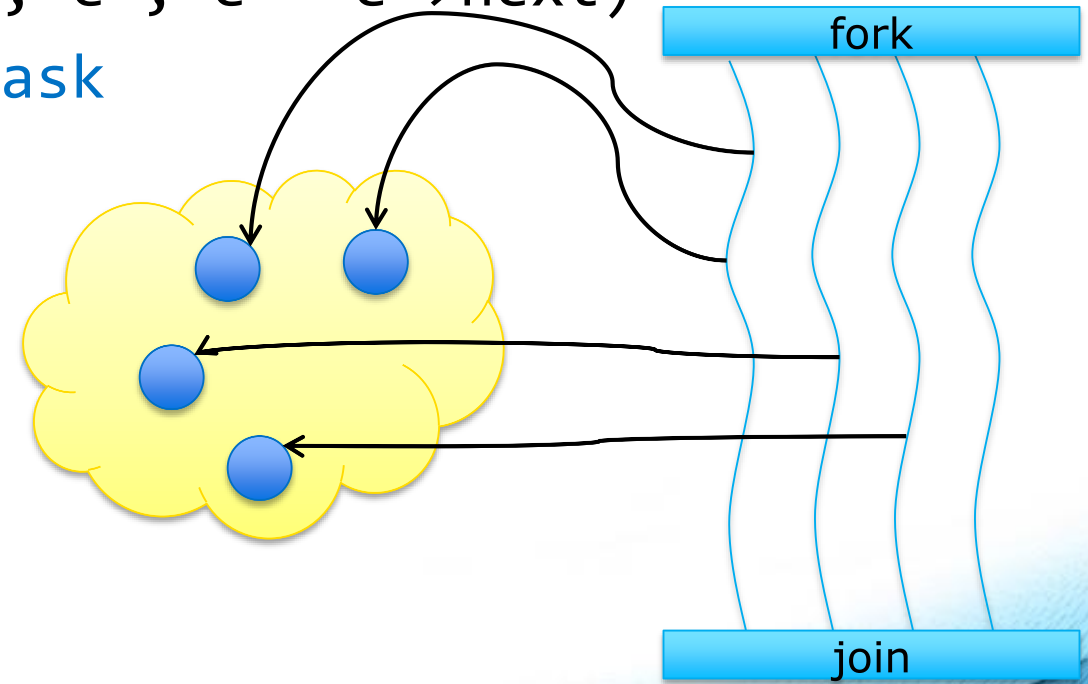
```
double a[N];
double l,s = 0;
#pragma omp parallel for reduction(+:s) private(l) \
                          schedule(static,4)

for (i = 0; i<N; i++)
{
   l = log(a[i]);
   s += l;
}
```



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);
```



The Present

OpenMP of the Present

Cluster	Group of computers communicating through fast intercon
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional unit
Vector	Single instruction using multiple functional units

OpenMP 4.0 for Devices

OpenMP 4.0 SIMD

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()`)

```
#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++)

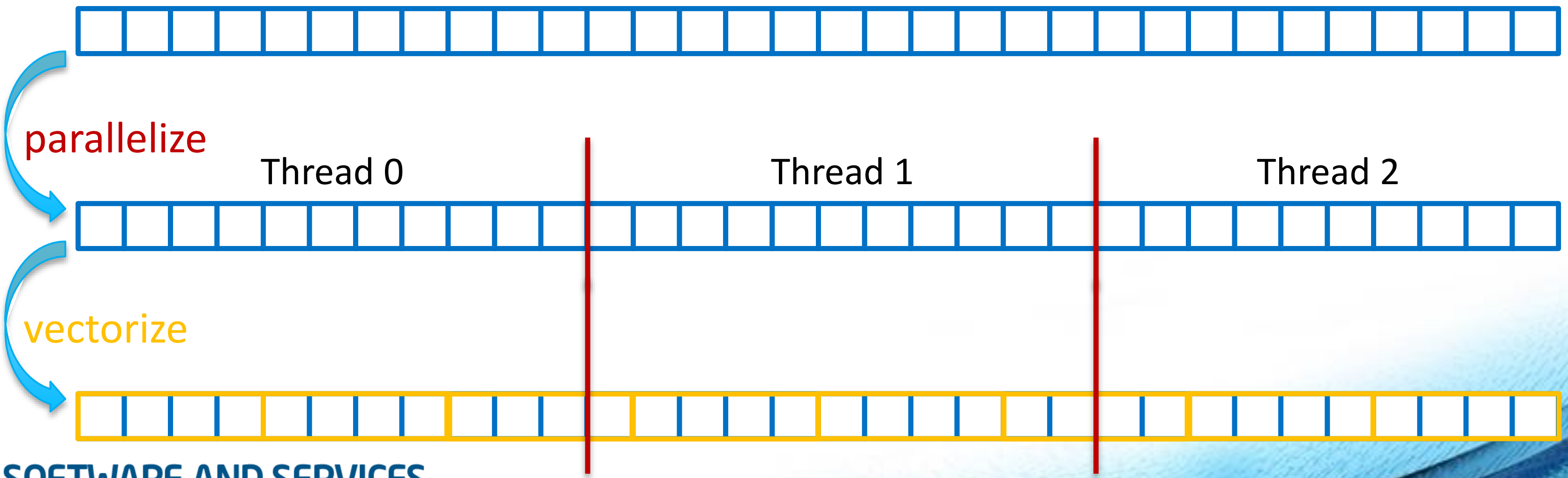
```
#pragma omp [for] simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp [do] simd [clause[[,] clause],…]
do-loops
```

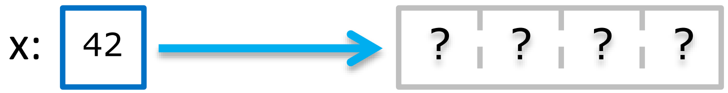
Example

```
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;
}
```

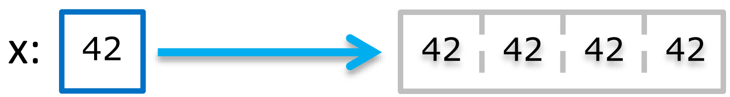


Data Sharing Clauses

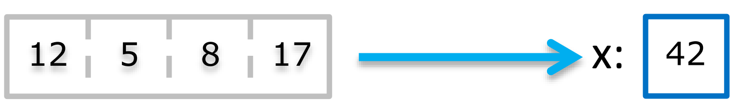
- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*



- `firstprivate(`*`var-list`*`):`
  Initialized vectors for variables in *var-list*



- `reduction(`*`op`*`:`*`var-list`*`):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



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

    $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (`*`list`*`[:`*`alignment`*`])`
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- `collapse (`*`n`*`)`

SIMD Function Vectorization

```
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

```
#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;
}
```

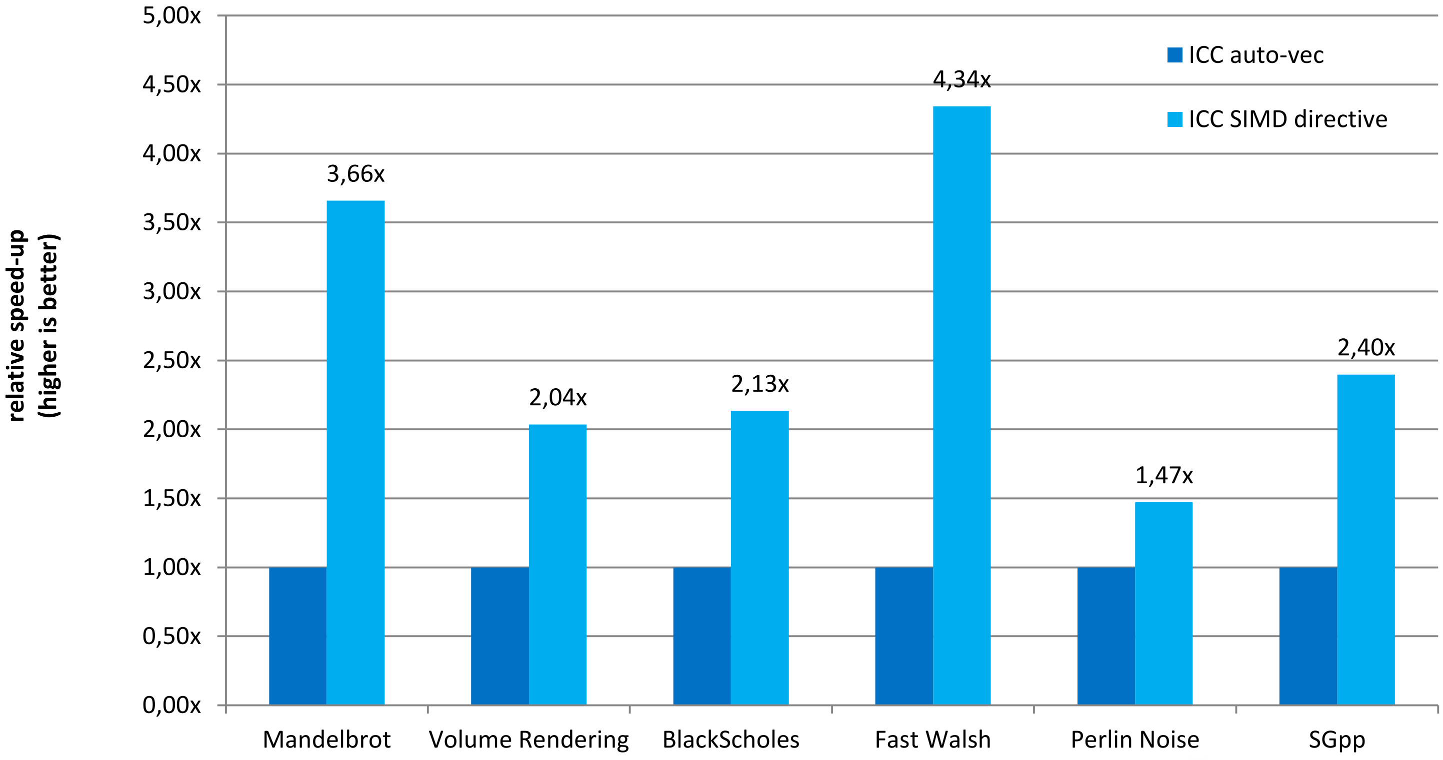
```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb, vc))
```

SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - generate function to support a given vector length
- `uniform (`*`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 (`*`argument-list[:linear-step]`*`)`
- `aligned (`*`argument-list[:alignment]`*`)`
- `reduction (`*`operator`*`:list)`

SIMD Constructs & Performance

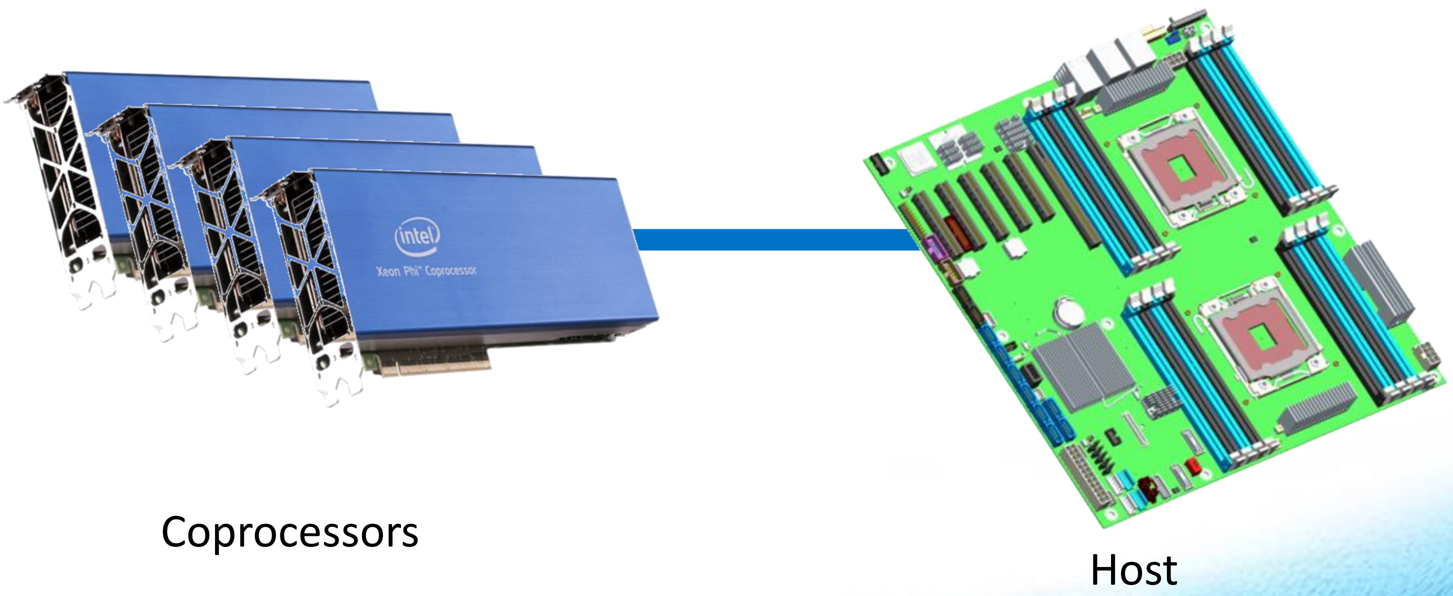


M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

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]
```

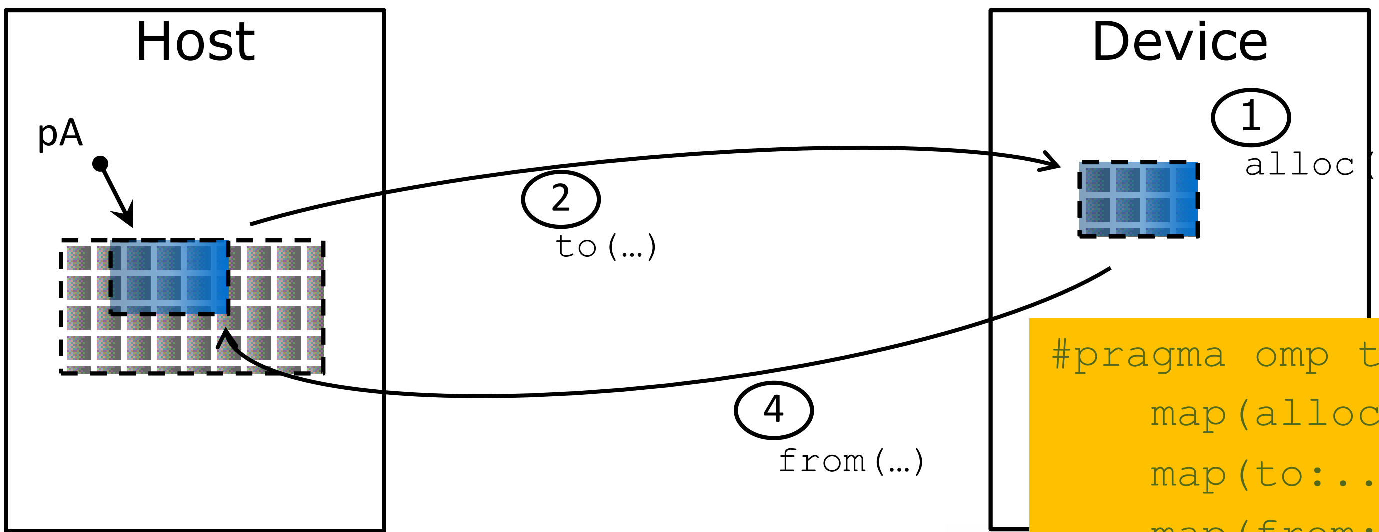
- Clauses
```
device(scalar-integer-expression)
map([alloc | to | from | tofrom:] list)
if(scalar-expr)
```

Execution Model

- The `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

Execution Model

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




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

3

Example

```
#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++):
```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

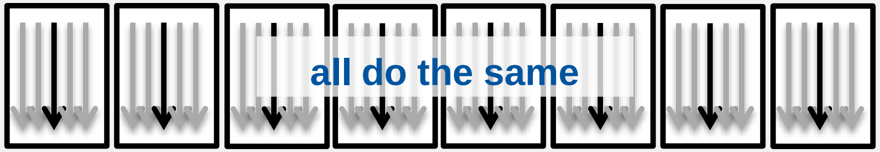
- Syntax (Fortran):
```
!$omp teams [clause[[,] clause],…]
structured-block
```

- 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

```
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)
```

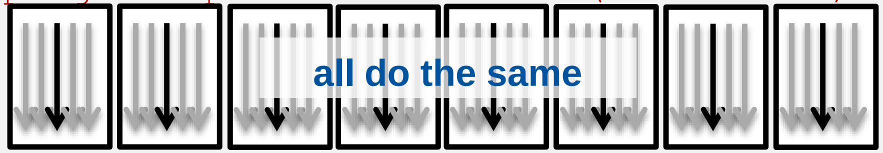



```
  for (int i = 0; i < n; i += num_blocks){
    for (int j = i; j < i + num_blocks; j++) {
        y[j] = a*x[j] + y[j];
  } }
  }
  free(x); free(y); return 0;
}
```

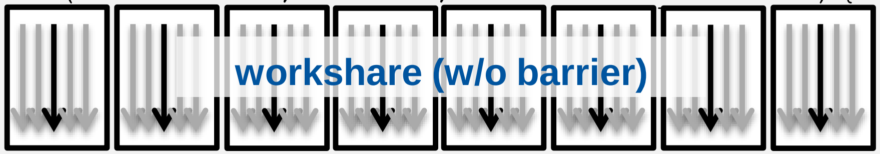
Offloading SAXPY to a Coprocessor

```
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

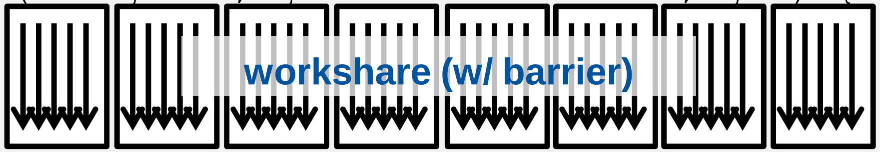
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```




```
#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

```
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 `dgemm` in MKL

```
void example() {
#pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // 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 `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

```
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) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

```
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;
    }
    // ...
}
```

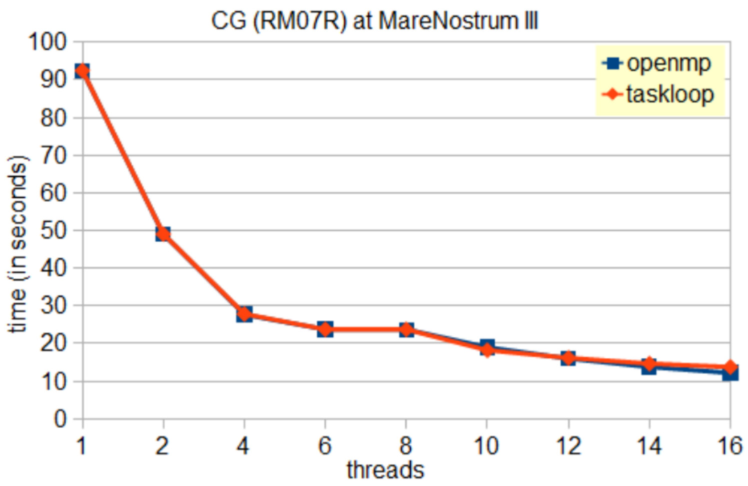
Example: Sparse CG

```
#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) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

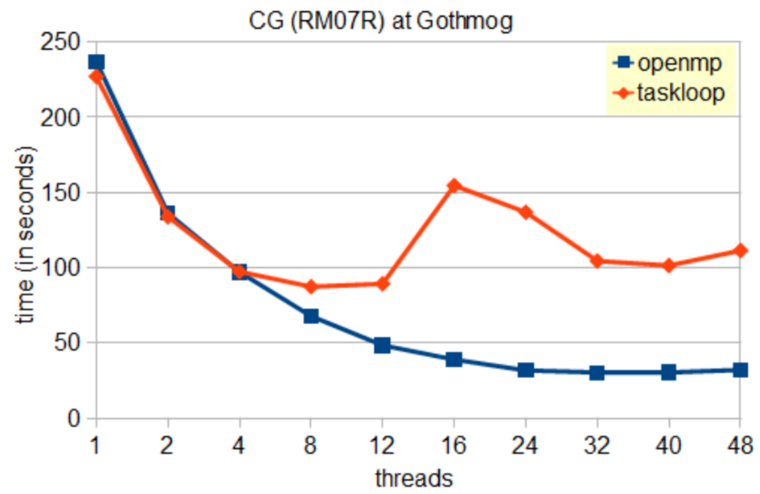
```
void matvec(Matrix *A, double *x, double *y) {
    // ...

#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)]
```

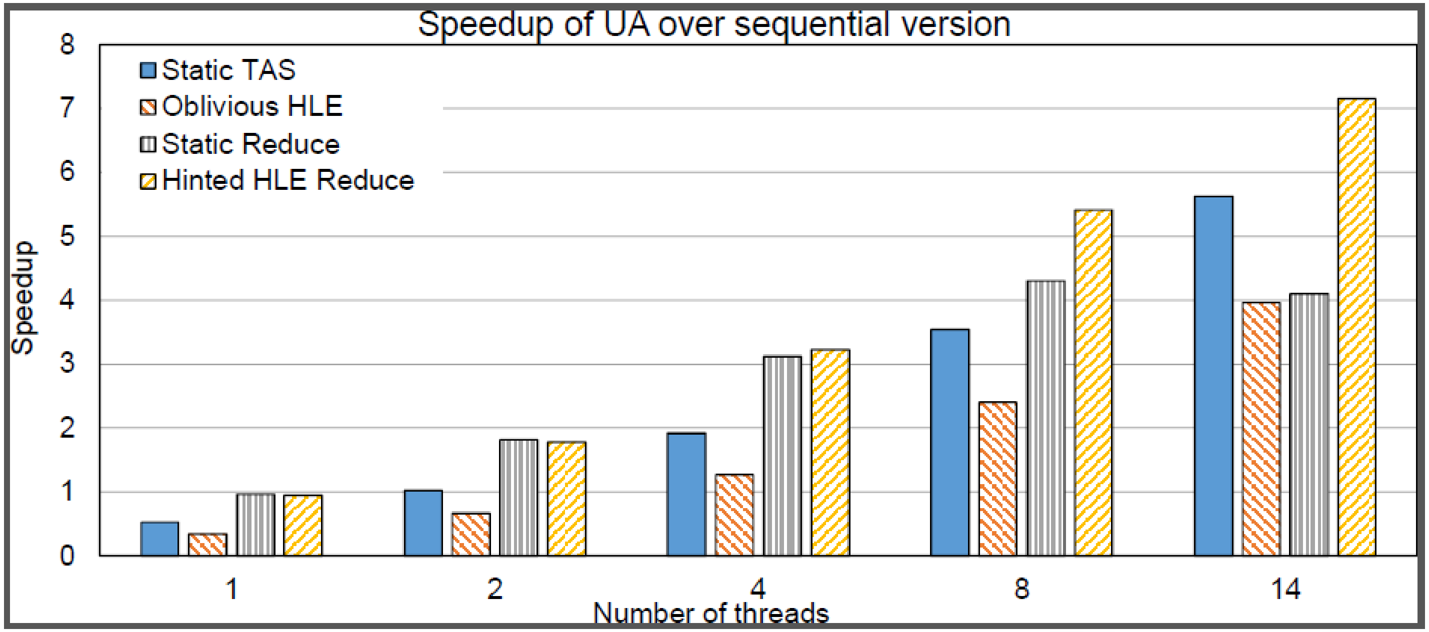
Examples

```
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_criticial() {
#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



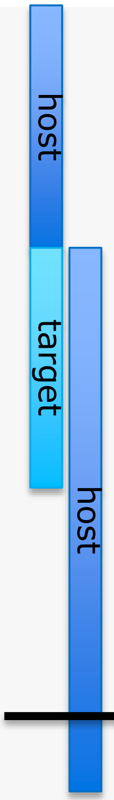
H. Bae, J.H. Cownie, M. Klemm, and C. Terboven. A User-guided Locking API for the OpenMP Application Program Interface. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, Using and Improving OpenMP for Devices, Tasks, and More, pages 173-186, Salvador, Brazil, September 2014. LNCS 8766.

Device Constructs

Asynchronous Offloading in 4.0

- You can this at your own risk ☺

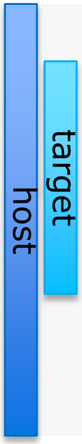
```
#pragma omp parallel sections num_threads(2)
{
#pragma omp task
  {
#pragma omp target map(to:input[:N]) map(from:result[:N])
#pragma omp parallel for
    for (i=0; i<N; i++) {
      result[i] = some_computation(input[i], i);
    }
  }
#pragma omp task
  {
    do_something_important_on_host();
 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



Asynchronous Offloading in 4.1

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

```
#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)
```

Creating and Destroying Device Data

```
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

intel®