Introduction to OpenMP

part III of III & Outlook on OpenMP 4.0 for Accelerators

Christian Terboven <terboven@rz.rwth-aachen.de>

31.07.2013 / Aachen, Germany

Stand: 22.07.2013

Version 2.3

Agenda

- **Avoiding Overhead: nowait, collapse, if, final and mergeable**
- **Iterator Loops and User-defined Reductions**
- **Task Scheduling and Task Dependencies**
- **Outlook: OpenMP for Accelerators**
  - What is an Accelerator in OpenMP?
  - Execution and Data Model
  - Target Construct
  - Example: SAXPY
  - Outlook: Asynchronicity
- **OpenMP 4.0 Feature Overview**

Avoiding Overhead

The nowait Clause

- **A worksharing construct (do/for, sections, single) has no barrier on entry – however, an implied barrier exists at the end of the worksharing region, unless the *nowait* clause is specified.**

- **Static schedule guarantees since OpenMP 3.0:**

```
#pragma omp for schedule(static) nowait
   for(i = 1; i < N; i++)
      a[i] = …
#pragma omp for schedule(static)
   for (i = 1; i < N; i++)
      c[i] = a[i] + …
```

Allowed in OpenMP 3.0 if and only if:
- Number of iterations is the same
- Chunk is the same (or not specified)

The collapse Clause

- **Loop collapsing: Ask the compiler to fuse perfectly nested loops to exploit a larger iteration space for the parallelization:**

```
#pragma omp for collapse(2)
    for(i = 1; i < N; i++)
        for(j = 1; j < M; j++)
            for(k = 1; k < K; k++)
                foo(i, j, k);
```

Iteration space from i-loop and j-loop is collapsed into a single one, if loops are perfectly nested and form a rectangular iteration space.

`if` Clause: Parallel Region

- **If the expression of an `if` clause on a *Parallel Region* evaluates to false**
  - The Parallel Region is executed with a Team of one Thread only

  → Used for optimization, e.g. avoid going parallel

- **OpenMP data scoping rules still apply!**

C/C++

```
#pragma omp parallel if(expr)
    ...
```

Fortran

```
!$omp parallel if(expr)
    ...
```

`if` Clause: Tasks

- **If the expression of an `if` clause on a *task* evaluates to false**
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when new tasks finishes

  → Used for optimization, e.g. avoid creation of small tasks

C/C++

```
#pragma omp task if(expr)
    ...
```

Fortran

```
!$omp task if(expr)
    ...
```

`final` Clause

- **For recursive problems that perform task decomposition, stop task creation at a certain depth exposes enough parallelism and reduces the overhead.**
  - final task: forces all child tasks to be final and included = execution is sequentially included in the task region (undeferred execution).

C/C++	Fortran
`#pragma omp task final(expr)`	`!$omp task final(expr)`

- **But: merging the data environment may have side-effects**

```
void foo(bool arg)
{
   int i = 3;
   #pragma omp task final(arg) firstprivate(i)
      i++;
   printf("%d\n", i);   // will print 3 or 4 depending on expr
}
```

`mergeable` Clause

- **If the mergeable clause is present, the implementation is allowed to merge the task‘s data environment with the enclosing region**
  - if the generated task is undeferred or included
    - undeferred: if clause present and evaluates to false
    - included: final clause present and evaluates to true

C/C++	Fortran
`#pragma omp task mergeable`	`!$omp task mergeable`

- **Personal Note: As of today (07/2013), no compiler or runtime implement final and/or mergeable in a way that-real world application may profit from using these clauses ☹.**

The taskyield Directive

- **The `taskyield` directive specifies that the current task can be suspended in favor of execution of a different task.**
  - Hint to the runtime for optimization and/or deadlock prevention

C/C++	Fortran
`#pragma omp taskyield`	`!$omp taskyield`

taskyield Example (1/2)

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
   for(int i = 0; i < n; i++)
      #pragma omp task
      {
         something_useful();
         while( !omp_test_lock(lock) ) {
            #pragma omp taskyield
         }
         something_critical();
         omp_unset_lock(lock);
      }
}
```

taskyield Example (2/2)

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
   for(int i = 0; i < n; i++)
      #pragma omp task
      {
         something_useful();
         while( !omp_test_lock(lock) ) {
            #pragma omp taskyield
         }
         something_critical();
         omp_unset_lock(lock);
      }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.

Iterator Loops and User-defined Reductions

Example: Bounding Box Code

- **This computes a bounding box of a 2D point cloud:**

```
struct Point2D;      /* data structure as you would expect it */
Point2D lb(RANGE, RANGE)        /* lower bound - init with max */
Point2D ub(0.0f, 0.0f);         /* upper bound - init with min */
for (std::vector<Point2D>::iterator it = points.begin();
                               it != points.end(); it++) {
   Point2D &p = *it;       /* compare every point to lb, ub*/
   lb.setX(std::min(lb.getX(), p.getX()));
   lb.setY(std::min(lb.getY(), p.getY()));
   ub.setX(std::max(ub.getX(), p.getX()));
   ub.setY(std::max(ub.getY(), p.getY()));
}
```

- **„Problems“ for an OpenMP parallelization?**
  - Reduction operation has to work with non-POD datatypes
  - Loop employs C++ iterator over std::vector datatype elements

Bounding Box w/ OpenMP 4.0

- **OpenMP 3.0 introduced Worksharing support for iterator loops**

```
#pragma omp for
      for (std::vector<Point2D>::iterator it =
          points.begin(); it != points.end(); it++) {
            ...
```

- **OpenMP 4.0 brings user-defined reductions**
  - *name*: `minp`, *datatype*: `Point2D`
  - *read*: `omp_in`, *written to*: `omp_out`, *initialization*: `omp_priv`

```
#pragma omp declare reduction(minp : Point2D :
    omp_out.setX(std::min(omp_in.getX(), omp_out.getX())),
    omp_out.setY(std::min(omp_in.getY(), omp_out.getY())) )
    initializer(omp_priv = Point2D(RANGE, RANGE))


#pragma omp parallel for reduction(minp:lb) reduction(maxp:ub)
      for (std::vector<Point2D>::iterator it =
          points.begin(); it != points.end(); it++) {
            ...
```

Task Scheduling and Task Dependencies

Tasks in OpenMP: Scheduling

- **Default: Tasks are *tied* to the thread that first executes them → not neccessarily the creator. Scheduling constraints:**
  - Only the thread a task is tied to can execute it
  - A task can only be suspended at a suspend point
    - Task creation, task finish, `taskwait`, `barrier`, `taskyield`
  - If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all tasks tied to the thread

- **Tasks created with the `untied` clause are never tied**
  - No scheduling restrictions, e.g. can be suspended at any point
  - But: More freedom to the implementation, e.g. load balancing

Unsafe use of `untied` Tasks

- **Problem: Because untied tasks may migrate between threads at any point, thread-centric constructs can yield unexpected results**

- **Remember when using `untied` tasks:**
  - Avoid `threadprivate` variables
  - Avoid any use of thread-ids (i.e. `omp_get_thread_num()`)
  - Be careful with `critical region` and *locks*

- **Simple Solution:**
  - Create a tied task region with

```
#pragma omp task if(0)
```

The depend Clause

C/C++

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

- **The *task dependence* is fulfilled when the predecessor task has completed**
  - `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - `out` and `inout` dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - The list items in a `depend` clause may include array sections.

Concurrent Execution w/ Dep.

- **Note: variables in the `depend` clause do not necessarily have to indicate the data flow**

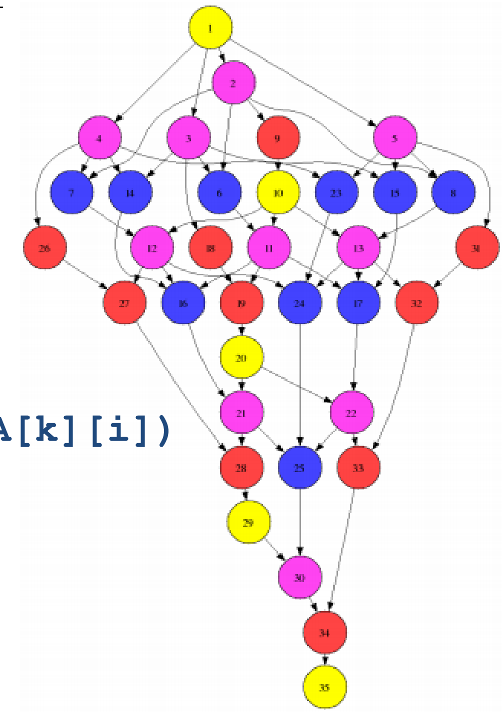
```
void process_in_parallel) {
   #pragma omp parallel
   #pragma omp single
   {
      int x = 1;
      ...
      for (int i = 0; i < T; ++i) {
         #pragma omp task shared(x, ...) depend(out: x)   // T1
            preprocess_some_data(...);
         #pragma omp task shared(x, ...) depend(in: x)    // T2
            do_something_with_data(...);
         #pragma omp task shared(x, ...) depend(in: x)    // T3
            do_something_independent_with_data(...);
      }
   } // end omp single, omp parallel
}
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

„Real“ Task Dependencies

```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
          spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
                sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
              ssyrk (A[k][i], A[i][i]);
       }
   }
}
```



Outlook: OpenMP for Accelerators

What is an Accelerator in OpenMP?

What kind of devices shall be supported?

- **In how differs an accelerator from just another core?**
  - different functionality, i.e. optimized for something special
  - different (possibly limited) instruction set

  → heterogeneous device

- **Assumptions used as design goals for OpenMP 4.0:**
  - every accelerator device is attached to one host device
  - it is probably heterogeneous
  - it may not be programmable in the same language as the host, or it may not implement all operations available on the host
  - it may or may not share memory with the host device
  - some accelerators are specialized for loop nests

Execution Model and Data Model

Execution Model

- **Host-centric: the execution of an OpenMP program starts on the *host device* and it may offload *target regions* to *target devices***
  - In principle, a target region also begins as a single thread of execution: when a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread/task [on the host] waits at the construct until the execution of the region completes

- **If a target device is not present, or not supported, or not available, the target region is executed by the host device**

- **If a construct creates a *data environment*, the data environment is created at the time the construct is encountered**

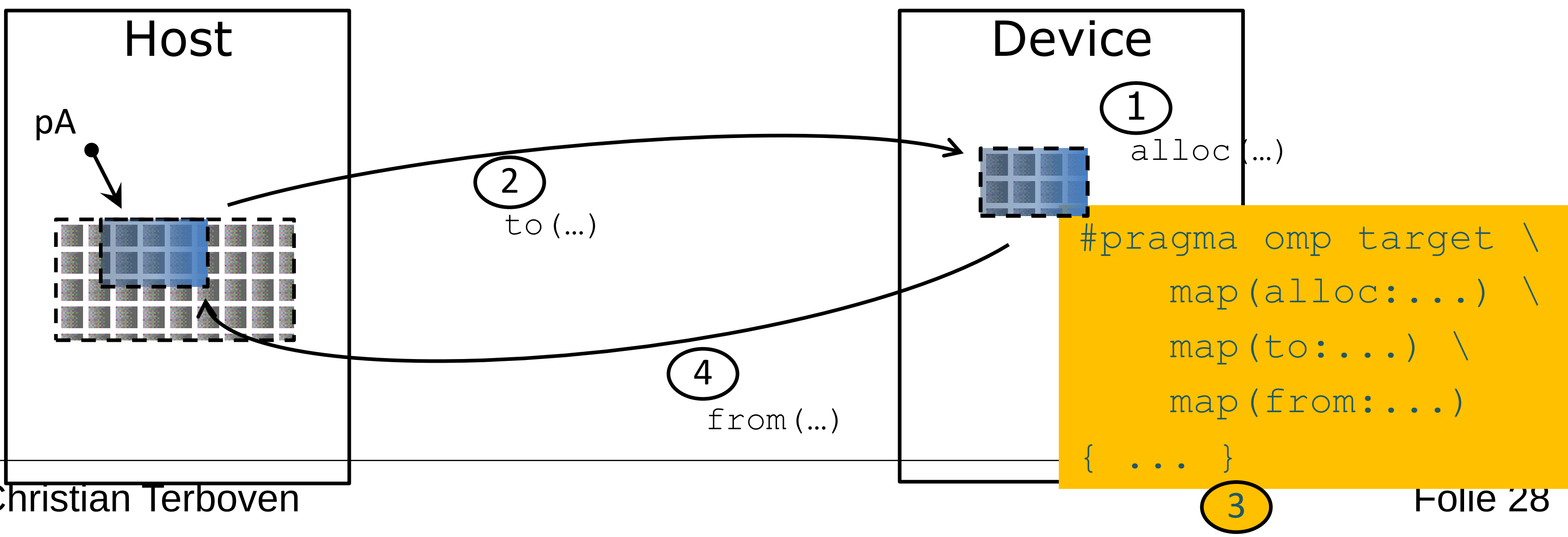
Data Model

- **When an OpenMP program begins, each device has an initial *device data environment***

- **Directives accepting data-mapping attribute clauses determine how an *original* variable is mapped to a *corresponding* variable in a device data environment**
  - original: the variable on the host
  - corresponding: the variable on the device
  - the corresponding variable in the device data environment may share storage with the original variable (danger of data races)

- **If a corresponding variable is present in the enclosing device data environment, the new device data environment inherits the corresponding variable from the enclosing device**

- **Data environment is lexically scoped**
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released
- **Use target construct to**
  - Transfer control from the host to the device
  - Establish a data environment (if not yet done)
  - Host thread waits until offloaded region completed



Example: Execution and Data Model

- **Environment Variable OMP_DEFAULT_DEVICE=<int>: sets the device number to use in target constructs**

```
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
   B[i] += sin(B[i]);
```

  - map variable B to device, then execute parallel region on device, works probably pretty well on Intel Xeon Phi

```
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
   #pragma omp parallel for
   for (b = i; b < i+num_blocks; b++)
      B[b] += sin(B[b]);
```

  - same as above, but code probably better optimized for NVIDIA GPGPUs

Comparing OpenMP with OpenACC

- **OpenMP 4.0 – for Intel Xeon Phi:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
   B[i] += sin(B[i]);
```

- **OpenMP 4.0 – for NVIDIA GPGPU:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
   #pragma omp parallel for
   for (b = i; b < i+num_blocks; b++)
      B[b] += sin(B[b]);
```

- **OpenACC – for NVIDIA GPGPU:**

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\
            vector_length(bsize)
#pragma acc loop gang vector
  for (i=0; i<N; ++i) {
    B[i] += sin(B[i]);
  }
```

Comparing OpenMP with OpenACC

- **OpenMP 4.0 – for Intel Xeon Phi:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
   B[i] += sin(B[i]);
```

- **OpenMP 4.0 – for NVIDIA GPGPU:**

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
   #pragma omp parallel for
   for (b = i; b < i+num_blocks; b++)
      B[b] += sin(B[b]);
```

- **OpenACC – for NVIDIA GPGPU:**

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\
            vector_length(bsize)
#pragma acc loop gang vector
  for (i=0; i<N; ++i) {
    B[i] += sin(B[i]);
  }
```

Changed to RC2:

Combined directive

#pragma omp teams distribute parallel for

Target Construct

target data construct

- **Creates a device data environment for the extent of the region**
  - when a target data construct is encountered, a new device data environment is created, and the encountering task executes the target data region
  - when an if clause is present and the if-expression evaluates to false, the device is the host

- **C/C++:**

The syntax of the **target data** construct is as follows:

#pragma omp target data *[clause[[,] clause],...] new-line*
structured-block

where *clause* is one of the following:

device(*integer-expression* **)**
map(*[map-type* **:** *] list* **)**
if(*scalar-expression* **)**

map clause

- **Map a variable from the current task's data environment to the device data environment associated with the construct**
  - the list items that appear in a map clause may include array sections
  - ***alloc***-type: each new corresponding list item has an undefined initial value
  - ***to***-type: each new corresponding list item is initialized with the original lit item's value
  - ***from***-type: declares that on exit from the region the corresponding list item's value is assigned to the original list item
  - ***tofrom***-type: the default, combination of to and from

- **C/C++:**

The syntax of the **map** clause is as follows:

map(*[map-type* **:** *] list* **)**

- **Creates a device data environment and execute the construct on the same device**
  - superset of the target data constructs - in addition, the target construct specifies that the region is executed by a device and the encountering task waits for the device to complete the target region

- **C/C++:**

The syntax of the **target** construct is as follows:

#pragma omp target *[clause[[,] clause],...] new-line*
structured-block

where *clause* is one of the following:

device(*integer-expression* **)**
map(*[map-type* **:** *] list* **)**
if(*scalar-expression* **)**

Example: Target Construct

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

```
#pragma omp target
#pragma omp teams num_teams(8) num_threads(4)
#pragma omp distribute
  for ( k = 0; k < NUM_K; k++ )
  {
    #pragma omp parallel for
    for ( j = 0; j < NUM_J; j++ )
    {
       ...
    }
  }
```

target update construct

- **Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses**

- **C/C++:**

The syntax of the **target update** construct is as follows:

#pragma omp target update *motion-clause[, clause[[,] clause],...] new-line*

where *motion-clause* is one of the following:

to(*list* **)**
from(*list* **)**

and where *clause* is one of the following:

device(*integer-expression* **)**
if(*scalar-expression* **)**

declare target directive

- **Specifies that [static] variables, functions (C, C++ and Fortran) and subroutines (Fortran) are mapped to a device**
  - if a list item is a function or subroutine then a device-specific version of the routines is created that can be called from a target region
  - if a list item is a variable then the original variable is mapped to a corresponding variable in the initial device data environment for all devices (if the variable is initialized it is mapped with the same value)
  - all declarations and definitions for a function must have a declare target directive

- **C/C++:**

The syntax of the **declare target** directive is as follows:

#pragma omp declare target *new-line*
declarations-definition-seq
#pragma omp end declare target *new-line*

teams construct (1/2)

- **Creates a league of thread teams where the master thread of each team executes the region**
  - the number of teams is determined by the num_teams clause, the number of threads in each team is determined by the num_threads clause, within a team region team numbers uniquely identify each team (omp_get_team_num())
  - once created, the number of teams remeinas constant for the duration of the teams region
- **The teams region is executed by the master thread of each team**
- **The threads other than the master thread to not begin execution until the master thread encounteres a parallel region**
- **Only the following constructs can be closely nested in the team region: distribute, parallel, parallel loop/for, parallel sections and parallel workshare**

teams construct (2/2)

- **A teams construct must be contained within a target construct, which must not contain any statements or directives outside of the teams construct**
- **After the teams have completed execution of the teams region, the encountering thread resumes execution of the enclosing target region**

- **C/C++:**

The syntax of the **teams** construct is as follows

#pragma omp teams *[clause[[,] clause],...] new-line*
structured-block

where *clause* is one of the following:

num_teams(*integer-expression* **)**
num_threads(*integer-expression* **)**
default(shared | none)
private(*list* **)**
firstprivate(*list* **)**
shared(*list* **)**
reduction(*operator* **:** *list* **)**

distribute construct

- **Specifies that the iteration of one or more loops will be executed by the thread teams, the iterations are distributed across the master threads of all teams**
  - there is no implicit barrier at the end of a distribute construct
  - a distribute construct must be closely nested in a teams region

- **C/C++:**

The syntax of the **distribute** construct is as follows:

#pragma omp distribute *[clause[[,] clause],...] new-line*
for-loops

Where *clause* is one of the following:

private(*list* **)**
firstprivate(*list* **)**
collapse(*n* **)**
dist_schedule(*kind[, chunk_size]* **)**

All associated for-loops must have the canonical form described in Section 2.5.

SAXPY

SAXPY: Serial (Host)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE







  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }




  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

SAXPY: OpenACC v2 (NVIDIA GPGPU)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc parallel copy(y[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

SAXPY: OpenMP 4.0 (NVIDIA GPGPU)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

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

Outlook: Asynchronicity

Example: Asynchronous Offload via Tasks

- **For asynchronous execution use the task construct and task dependencies:**

```
#pragma omp target data map(alloc:Z)
{
   #pragma omp parallel for
   for (c = 0; c < nchunks; c += chunksz)
   {
      #pragma omp task dep(out:c)
      #pragma omp target update map(to: Z[c:chunksz])

      #pragma omp task dep(in:c)
      #pragma omp target
      #pragma omp parallel for
      for (i = c; i < c + chunksz; i++)
         Z[i] = f(Z[i]);
   }
}
```

OpenMP 4.0 Feature Overview

OpenMP 4.0 Achievements

- **End of a long road? A brief rest stop along the way…**

- **Addressed several major open issues for OpenMP**
- **Did not break existing code unnecessarily**
- **Included 103 passed tickets**
  - Focus on major tickets initially
  - Builds on two comment drafts (“RC1” and “RC2”)
  - Many small tickets after RC2, a few large ones

- **Final vote was held on July 11**

- **Already starting work on OpenMP 5.0**

Overview of major 4.0 additions

- **Covered previously**
  - Device constructs
  - Task dependences and task groups
  - Thread affinity control
  - Support for array sections (including in C and C++)
  - User-defined reductions
- **Not covered during this week**
  - SIMD constructs
  - Cancellation
  - Initial support for Fortran 2003
  - Sequentially consistent atomics
  - Display of initial OpenMP internal control variables

Thank you for your attention.