IWOMP 2016 Tutorial: OpenMP Accelerator Model

(OpenMP for Heterogeneous Computing)

James Beyer
NVIDIA

Bronis R. de Supinski
Lawrence Livermore National Laboratory

LLNL-PRES-704605

IWOMP 2016 Tutorial: OpenMP Accelerator Model

(OpenMP for Heterogeneous Computing)

Tom Scogland **Bronis R. de Supinski**

Lawrence Livermore National Laboratory

What is OpenMP?

- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Consists of compiler directives, runtime routines and environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
- ARB mission statement:

  "The OpenMP ARB mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable."
- **Version 4.5 was released in November 2015**

openmp.org/wp/about-openmp/

IWOMP 2016
12th Intl. Workshop on OpenMP
5-7 OCT | NARA, JAPAN

»*Public OpenMP Calendar*

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Follow @OpenMP_ARB

Search OpenMP.org
Google™ Custom Search
Search

Archives
- August 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- February 2016
- December 2015
- November 2015

Members

Permanent Members of the ARB:

- **AMD** (Greg Stoner)
- **ARM** (Chris Adeniyi-Jones)
- **Cray** (Luiz DeRose)
- **Fujitsu** (Eiji Yamanaka)
- **HP** (Sujoy Saraswati)
- **IBM** (Kelvin Li)
- **Intel** (Xinmin Tian)
- **Micron** (Kirby Collins)
- **NEC** (Kazuhiro Kusano)
- **NVIDIA** (Jeff Larkin)
- **Oracle Corporation** (Nawal Copty)
- **Red Hat** (Matt Newsome)
- **Texas Instruments** (Eric Stotzer)

Auxiliary Members of the ARB:

- **Argonne National Laboratory** (Kalyan Kumaran)
- **ASC/Lawrence Livermore National Laboratory** (Bronis R. de Supinski)
- **Barcelona Supercomputing Center** (Xavier Martorell)
- **Bristol University** (Simon McIntosh-Smith)
- **cOMPunity** (Barbara Chapman/Yonghong Yan)
- **Edinburgh Parallel Computing Centre (EPCC)** (Mark Bull)
- **INRIA** (Olivier Aumage)
- **Los Alamos National Laboratory** (David Montoya)
- **Lawrence Berkeley National Laboratory** (Alice Koniges/Helen He)
- **NASA** (Henry Jin)
- **Oak Ridge National Laboratory** (Oscar Hernandez)
- **RWTH Aachen University** (Dieter an Mey)
- **Sandia National Laboratory** (Stephen Olivier)
- **Texas Advanced Computing Center** (Kent Milfeld)
- **University of Houston** (Deepak Eachempati/Jeremy Kemp)

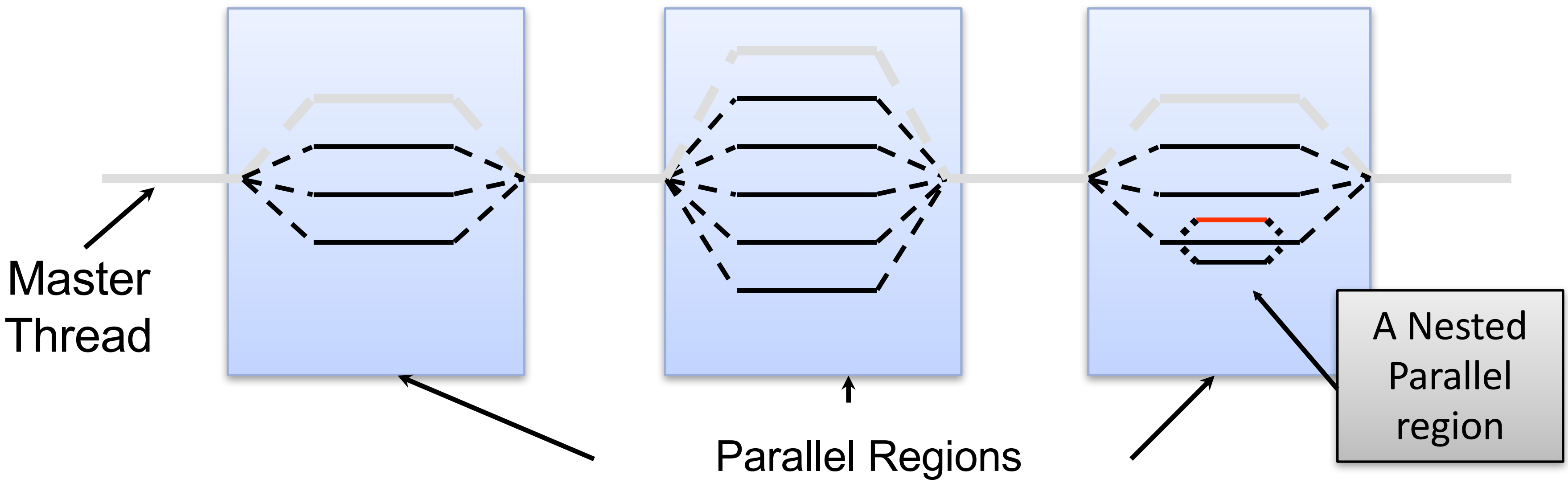
OpenMP is widely supported by the industry, as well as the academic community

OpenMP Before Version 4.0

OpenMP Execution Model

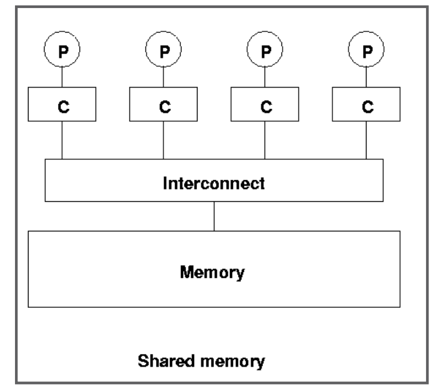
Before Device Directives

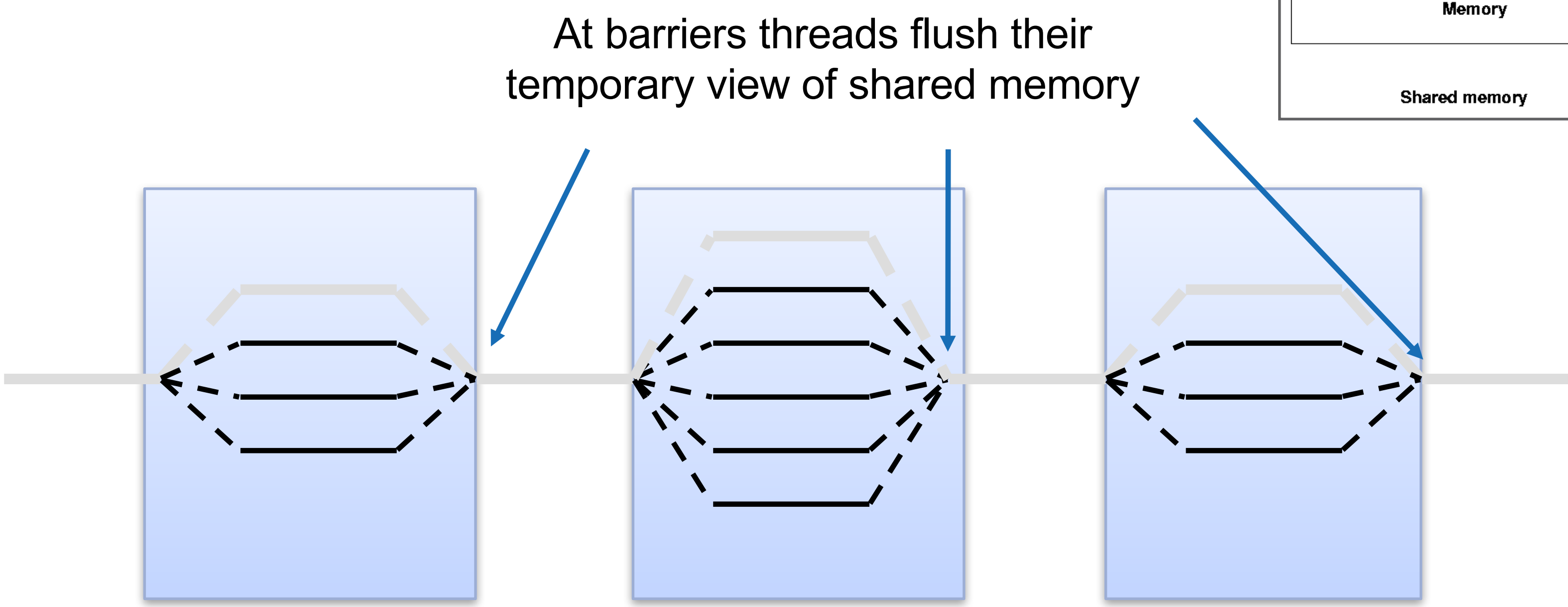
- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally until desired performance is achieved: i.e., the sequential program evolves into a parallel program.



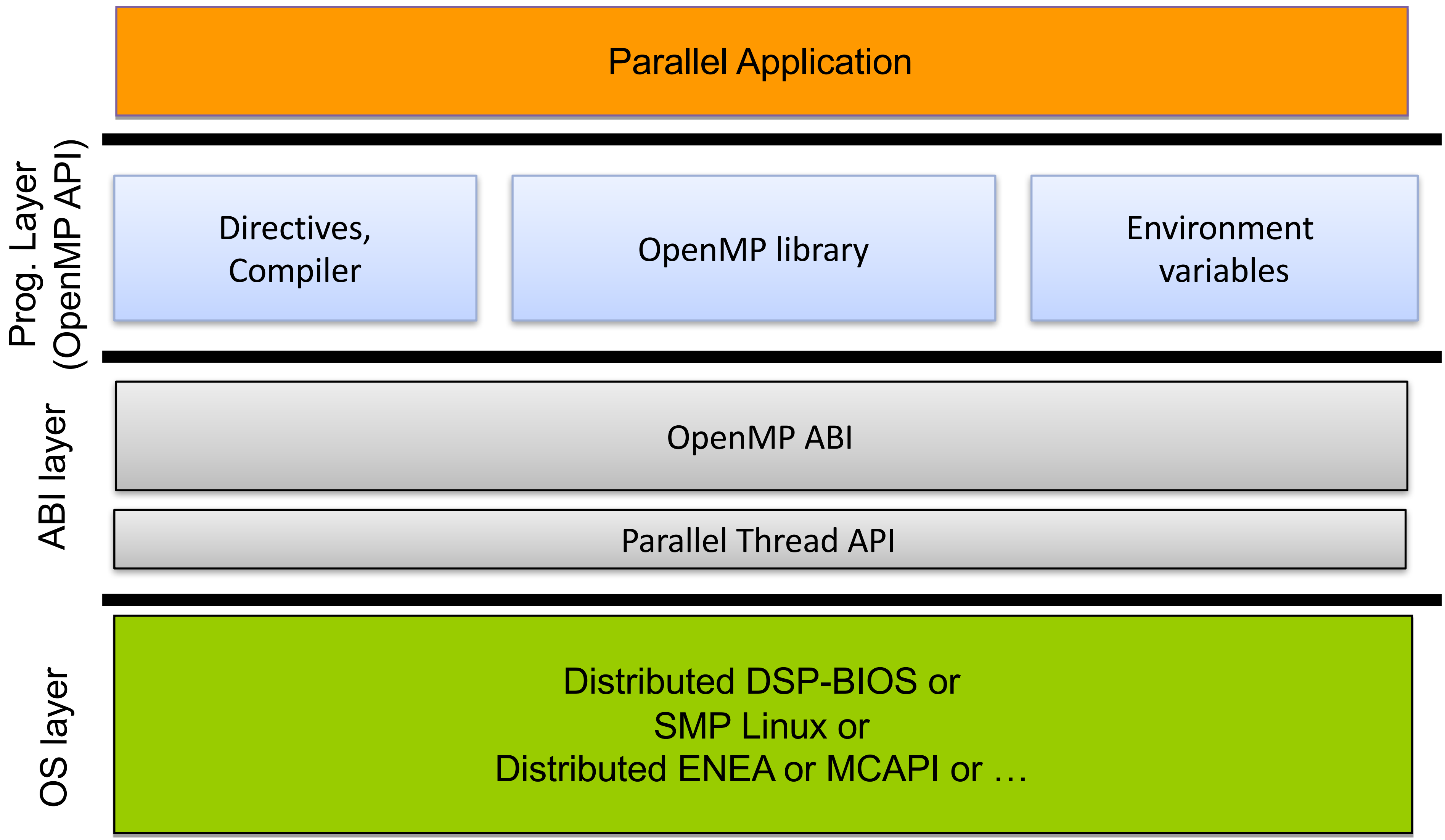
OpenMP Memory Model

- Threads have access to a *shared* memory
  - for **shared** data
  - each thread can have a temporary view of the shared memory (e.g., registers, cache) between synchronization barriers.
- Threads have *private* memory
  - for **private** data
  - Each thread has a stack for data local to each task it executes





OpenMP Parallel Computing Solution Stack



OpenMP Features

Provides the means to:

- create and destroy threads
- assign / distribute work (tasks) to threads
- specify which data is shared and which is private
- coordinate thread access to shared data

Syntax and Usage:

- Directives in OpenMP are compiler pragmas (usually) applied to a statement:
  #pragma omp *construct [clause [clause]…]*
  statement;
- An Include file:
  #include <omp.h>
- A runtime library:
  -l libomp.lib

OpenMP: Parallel Regions

- You create threads in OpenMP with the “omp parallel” pragma.
- For example, To create a 4-thread Parallel region:

Each thread redundantly executes the code within the structured block

```
float A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    lots_of_work(id,A);
}
```

- Each thread calls lots_of_work(id,A) for **id** = **0** to **3**

OpenMP Data Attributes: Private Clause

- Private(var) creates a local copy of var for each thread.
  – The value is uninitialized
  – Private copy is *not* storage associated with the original

```
void wrong(){
   int IS = 0;
#pragma parallel for private(IS)
   for(int J=1;J<1000;J++)
      IS = IS + J;
   printf("%i", IS);
}
```

OpenMP Parallel API

Compiler Directives

- Parallelization
  - parallel
- Worksharing
  - for ,sections, parallel for, task…
- Synchronization
  - barrier, critical, atomic, flush, …
- Data-sharing attributes
  - shared, private, firstprivate, threadprivate, reduction, …

Library Functions

- Thread Control
  - omp_get_thread_num(), omp_get_num_threads(), omp_set_num_threads(), …
- Locks
  - omp_set_lock(), omp_unset_lock(), …

Enviroment Variables

- **OMP_NUM_THREADS, OMP_SCHEDULE, …**

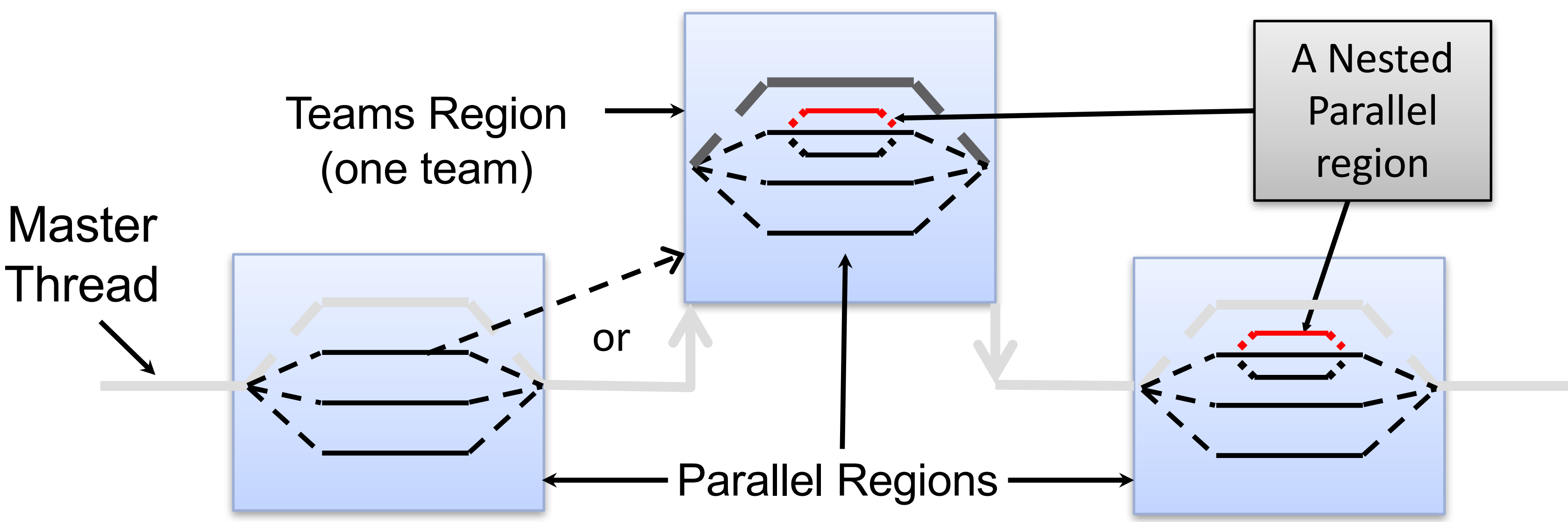
Summary

- Parallel programming model
  - Data parallelism (omp parallel for)
  - Task parallelism (omp task)
  - Productivity and flexibility
- Runtime requirements
  - Thread create/destroy on multiple cores
  - Barriers
  - Locks (semaphores, atomics, mutex, …)
  - Shared/private memory management
  - Memory consistency model

OpenMP as of Version 4.0

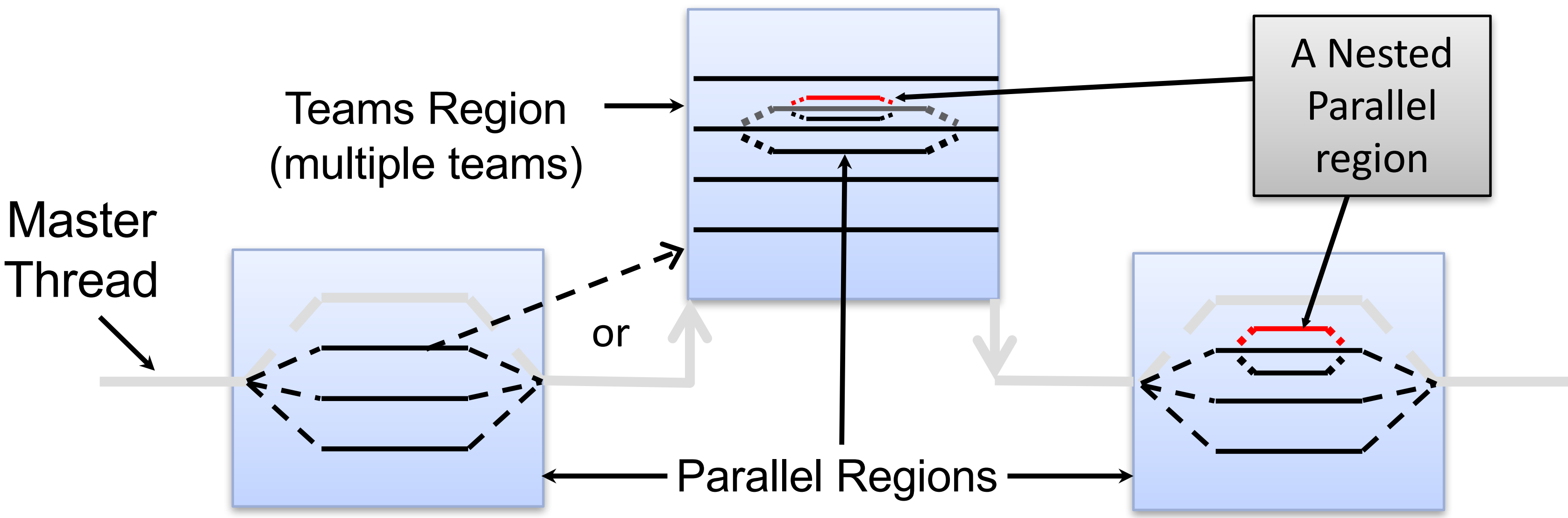
OpenMP Execution Model

- Master thread spawns a team of threads as needed.
- Threads spawn leagues of thread teams as needed.
- Parallelism is added incrementally until desired performance is achieved: i.e., the sequential program evolves into a parallel program.



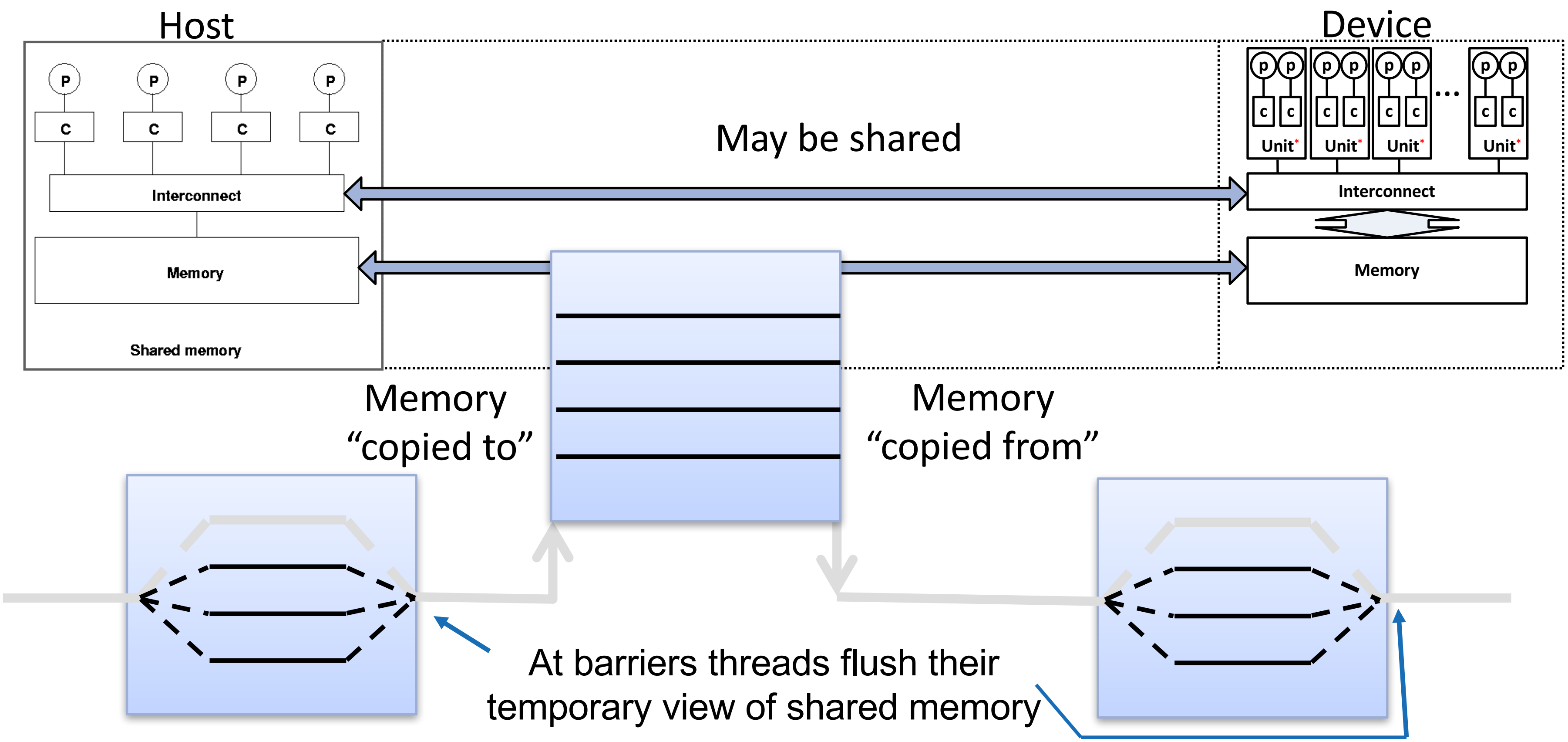
OpenMP Execution Model

- Master thread spawns a team of threads as needed.
- Threads spawn leagues of thread teams as needed.
- Parallelism is added incrementally until desired performance is achieved: i.e., the sequential program evolves into a parallel program.



OpenMP Memory Model

- Threads have access to a *shared* memory
- Threads have *private* memory
- Devices may have distinct memory or they may share memory with the "host"



OpenMP Features

Provides the means to:

- create and destroy threads
- create and destroy leagues of thread teams
- assign / distribute work (tasks) to threads and devices
- specify which data is shared and which is private
- specify which data must be available to the device
- coordinate thread access to shared data

Syntax and Usage:

- Directives in OpenMP are compiler pragmas applied to a statement:
  #pragma omp *construct [clause [clause]…]*
  statement;
- An Include file:
  #include <omp.h>
- A runtime library:
  -l libomp.lib

OpenMP: Target Regions

- You "offload" work in OpenMP with the "omp target" pragma.
- For example, To create a Target region:

A new master thread executes the code within the structured block

```
float A[1000];
#pragma omp target
{
   int tid = omp_get_team_num();
   lots_of_work(tid,A);
}
```

OpenMP: Target Teams Regions

- You "offload" work in OpenMP to multiple "workers" with the "omp target teams" pragma.
- For example, To create a 32 member Target Teams region :

Each new master thread executes the code within the structured block

```
float A[1000];
#pragma omp target teams num_teams(32)
{
   int tid = omp_get_team_num();
   lots_of_work(tid,A);
}
```

OpenMP: Target Data Regions

- You make data available on the device the the “omp target data” pragma
- For example, To make ‘A’ available on the device:

In the case of distinct memory the object A is copied to the device at the beginning of the block and copied back to the host at the end of the block.

```
float A[1000];
#pragma omp target data map(A[0:1000])
{
// A "belongs" to the device here
}
```

IWOMP 2016 Tutorial: OpenMP Accelerator Model

(OpenMP for Heterogeneous Computing)

Tom Scogland **Bronis R. de Supinski**

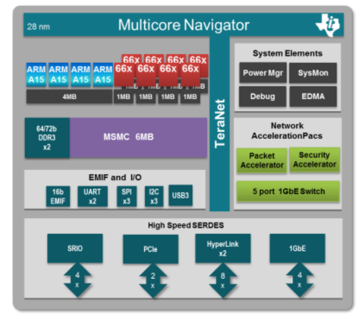
Lawrence Livermore National Laboratory

Topics

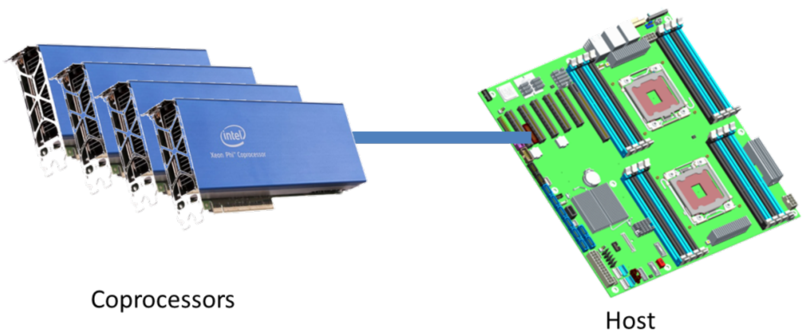
- Heterogeneous device execution model
- Mapping variables to a device

Device Model

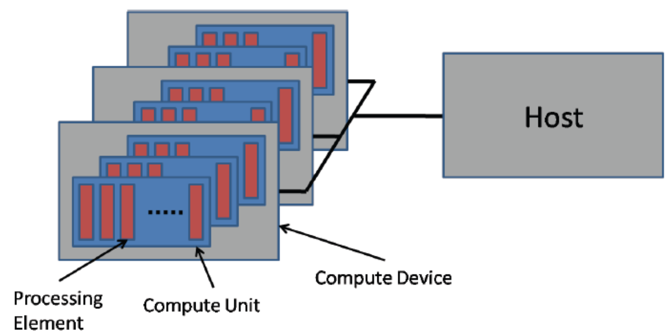
- OpenMP 4.0+ supports heterogeneous systems
- Device model:
  - →One host device and
  - →One or more target devices



Heterogeneous SoC



Host and Co-processors



Host and GPUs

Terminology

- Device:
  an implementation-defined (logical) execution unit
- Device data environment:
  The storage associated with a device.

The execution model is host-centric such that the host device (the initial device) offloads **target** regions to target devices.

OpenMP Device Constructs

- Execute code on a target device
  - **omp target** *[clause[[,] clause],…]*
    structured-block
  - **omp declare target**
    [function-definitions-or-declarations]
- Manage the device data environment
  - **map** *([map-type:] list) // map clause*
    map-type := ***alloc*** | ***tofrom*** | ***to*** | ***from***
  - **omp target data** *[clause[[,] clause],…]*
    structured-block
  - **omp target update** *[clause[[,] clause],…]*
  - **omp declare target**
    [variable-definitions-or-declarations]
- Workshare for acceleration
  - **omp teams** *[clause[[,] clause],…]*
    structured-block
  - **omp distribute** *[clause[[,] clause],…]*
    for-loops

Device Runtime Support

- Runtime support routines
  - `void omp_set_default_device(int dev_num )`
  - `int  omp_get_default_device(void)`
  - `int  omp_get_num_devices(void);`
  - `int  omp_get_num_teams(void)`
  - `int  omp_get_team_num(void);`
  - `int  omp_is_initial_device(void);`
  - `int  omp_get_initial_device(void);`
- Environment variable
  - Control default device through **`OMP_DEFAULT_DEVICE`**
    - Accepts a non-negative integer value
  - Control default contention group size **`OMP_THREAD_LIMIT`**

Offloading Computation

- Use **target** construct to
  - → Transfer control from the host to the target device
  - → Map variables between the host and target device data environments
- Host thread waits until offloaded region completed
- Use **`nowait`** for asynchronous execution

```
#pragma omp target map(to:b,c,d) map(from:a)
  {
#pragma omp parallel for
    for (i=0; i<count; i++) {
      a[i] = b[i] * c + d;
    }
  }
```

`target` Construct

- Transfer control from the host to the device
- Syntax (C/C++)

  ```
  #pragma omp target [clause[[,] clause],…]
  structured-block
  ```
- Syntax (Fortran)

  ```
  !$omp target [clause[[,] clause],…]
  structured-block
  !$omp end target
  ```
- Clauses
  - → List grew in OpenMP 4.5
  - → More will be added in OpenMP 5.0 TR

Target construct clauses

```
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if([target:]scalar-expr)
private(list)
firstprivate(list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

map Clause

```c
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target map(v1[0:N], v2[:N], p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];

   output(p, N);
}
```

```fortran
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer :: N,i
   call init(v1, v2, N)
   !$omp target map(v1(1:N), v2(:N), p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !omp end target
   call output(p, N)
end subroutine
end module
```

- The array sections for v1, v2, and p are explicitly *mapped* into the device data environment.
- The variable N is implicitly *firstprivate*

map Clause (differences)

```
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target map(v1, v2, p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];

   output(p, N);
}
```

```
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer :: N,i
   call init(v1, v2, N)
   !$omp target map(v1, v2, p)
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !omp end target
   call output(p, N)
end subroutine
end module
```

C/C++

- The “arrays” v1, v2, and p are not explicitly mapped! The “scalars” v1, v2, and p are explicitly mapped

Fortran

- The array sections for v1, v2, and p are explicitly *mapped* into the device data environment.

Both

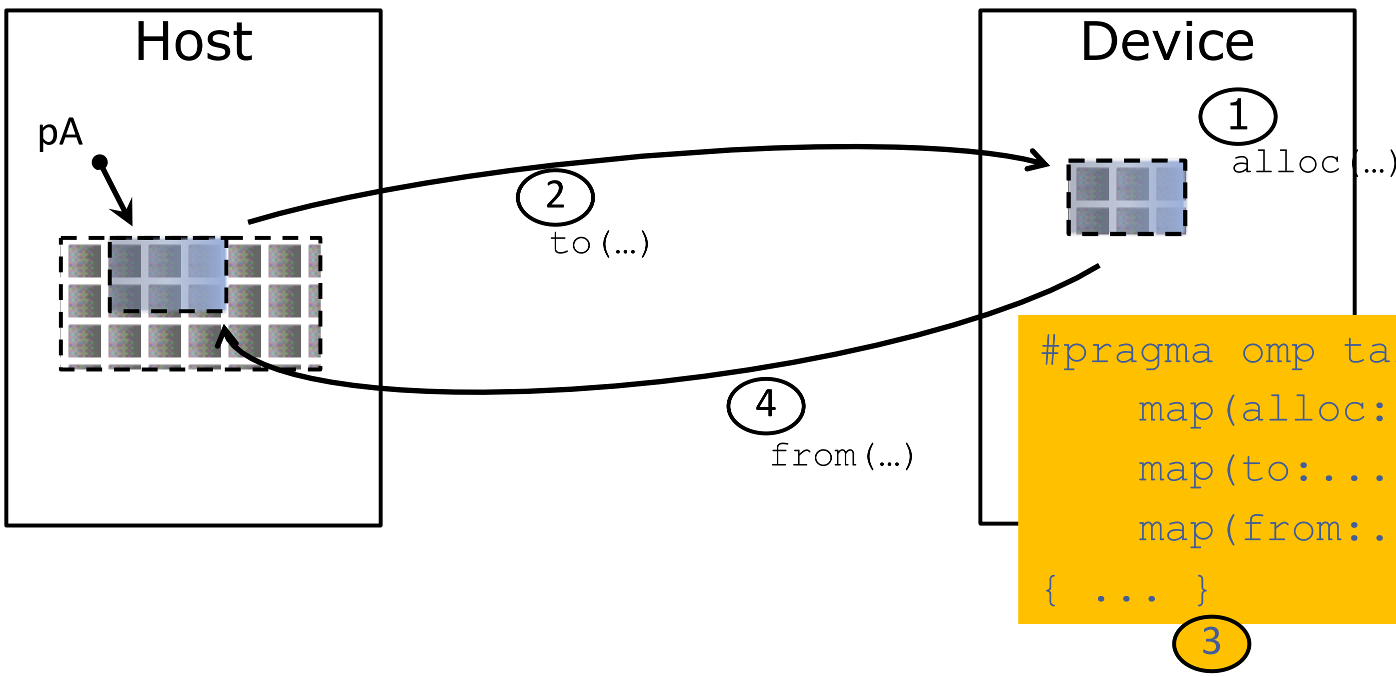
- The variable N is implicitly *firstprivate*

Terminology

- Mapped variable:
  An *original variable* in a (host) data environment has a *corresponding variable* in a device data environment
- Mappable type:
  A type that is amenable for mapped variables.
  (Bitwise copyable plus additional restrictions.)

Device Data Environment

- The **`map`** clauses determine how an *original variable* in a data environment is mapped to a *corresponding variable* in a device data environment.




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

`map` Clause

- Map a variable or an array section to a device data environment
- Syntax:

  `map([[`*`map-type-modifier[,]]map-type:] list`*`)`
- Where map-type is:
  - → `alloc:` allocate storage for corresponding variable
  - → `to:` allocate and assign value of original variable to corresponding variable on entry
  - → `from:` allocate and assign value of corresponding variable to original variable on exit
  - → `tofrom:` default, both to and form
  - → `release:` decrement list item's reference count by one
  - → `delete:` set list item's reference count to zero

`map` Clause

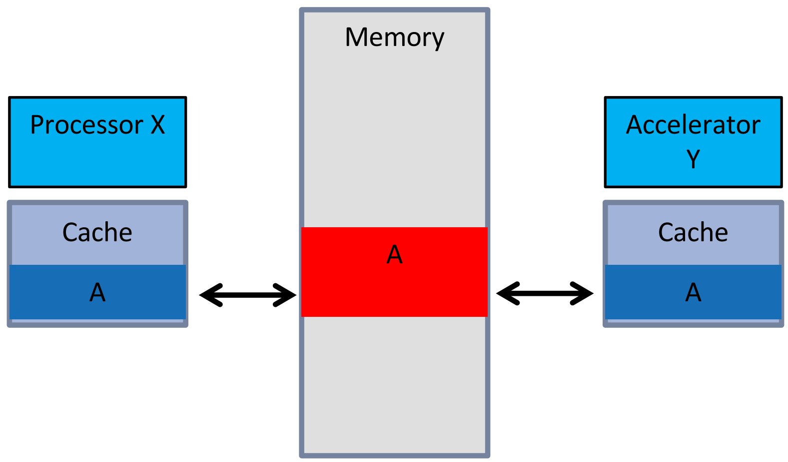
```
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target map(to:v1[0:N],v2[:N]) map(from:p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];

   output(p, N);
}
```

```
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer :: N,i
   call init(v1, v2, N)
   !$omp target map(to: v1(1:N), v2(:N)) map(from:p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !omp end target
   call output(p, N)
end subroutine
end module
```

- On entry to the target region:
  - Allocate corresponding variables v1, v2, and p in the device data environment.
  - Assign the corresponding variables v1 and v2 the value of their respective original variables.
  - The corresponding variable p is undefined.
- On exit from the target region:
  - Assign the original variable p the value of its corresponding variable.
  - The original variables v1 and v2 are undefined.
  - Remove the corresponding variables v1, v2, and p from the device data environment

MAP is not necessarily a copy

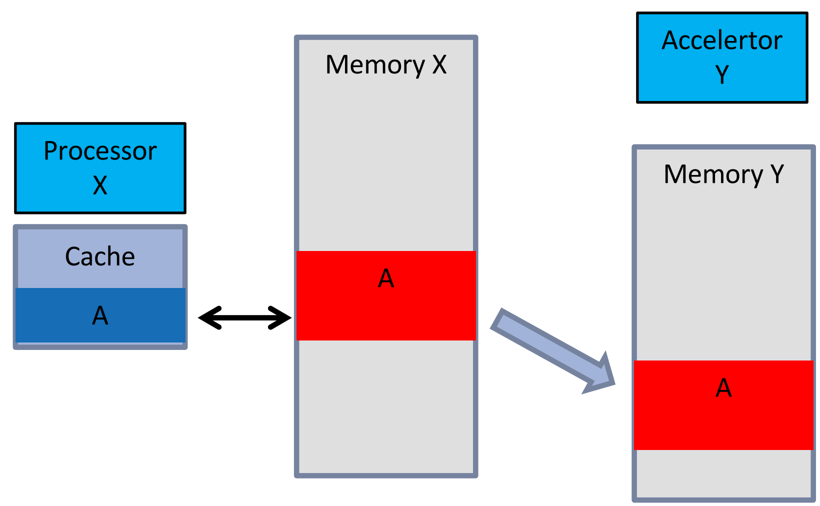
Shared memory



- The corresponding variable in the device data environment *may* share storage with the original variable.

- Writes to the corresponding variable may alter the value of the original variable.

Distributed memory



Map variables across multiple target regions

- Optimize sharing data between host and device
- The `target data` construct maps variables but does not offload code.
- Corresponding variables remain in the device data environment for the extent of the target data region
- Useful to map variables across multiple target regions
- The `target update` synchronizes an original variable with its corresponding variable.

`target data` Construct Example

```
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;

   init(v1, v2, N);

   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
          p[i] = v1[i] * v2[i];

      init_again(v1, v2, N);

      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
          p[i] = p[i] + (v1[i] * v2[i]);
   }

   output(p, N);
}
```

- The `target data` construct maps variables to the *device data environment*.
- v1 and v2 are mapped at each `target` construct.
- p is mapped once by the `target data` construct.

target [enter|exit]data Construct Example

```
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;

   init(v1, v2, N);

   #pragma omp target enter data map(alloc: p[0:N])
     #pragma omp target map(to: v1[:N], v2[:N])
     #pragma omp parallel for
     for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];

     init_again(v1, v2, N);

     #pragma omp target map(to: v1[:N], v2[:N])
     #pragma omp parallel for
     for (i=0; i<N; i++)
         p[i] = p[i] + (v1[i] * v2[i]);
   #pragma omp target exit data map(from: p[0:N])

   output(p, N);
}
```

- The `target enter data` construct maps variables to the *device data environment*.
- v1 and v2 are mapped at each `target` construct.
- p is mapped with the `target enter|exit data` construct.
- The `target exit data` construct maps the brings the variables back to the *host data environment*

Map variables to a device data environment

- The host thread executes the data region
- Be careful when using the device clause

```
#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);

    do_some_other_stuff_on_host();

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

`target data` Construct

- Map variables to a device data environment for the extent of the region.
- 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)
use_device_ptr(list)
```

`target enter data` Construct

- Map variables to a device data environment until it is removed.
- Syntax (C/C++)

  ```
  #pragma omp target enter data [clause[[,] clause],…]
  ```

  Syntax (Fortran)

  ```
  !$omp target enter data [clause[[,] clause],…]
  ```

  Clauses

  ```
  device(scalar-integer-expression)
  map(alloc | to : list)
  if([target enter data :]scalar-expr)
  depend(dependence-type : list)
  nowait
  ```

target exit data Construct

- Unmap variables from a device data environment.
- Syntax (C/C++)
  `#pragma omp target exit data [clause[[,] clause],…]`
  Syntax (Fortran)
  `!$omp target exit data [clause[[,] clause],…]`
- Clauses
  `device(scalar-integer-expression)`
  `map(from | release | delete: list)`
  `if([target exit data :]scalar-expr)`
  `depend(dependence-type : list)`
  `nowait`

Synchronize mapped variables

- Synchronize the value of an original variable in a host data environment with a corresponding variable in a device data environment

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target
#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 to(input[:N])

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

`target update` Construct

- Issue data transfers between host and devices
- Syntax (C/C++)

  `#pragma omp target update` *`[clause[[,] clause],…]`*

- Syntax (Fortran)

  `!$omp target update` *`[clause[[,] clause],…]`*

- Clauses

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

  `to(`*`list`*`)`

  `from(`*`list`*`)`

  `if(`*`scalar-expr`*`)`

  `nowait`

  `depend(`*`dependence-type: list`*`)`

Map a variable for the whole program

```
define N 1000
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target

extern void init(float *, float *, int);
extern void output(float *, int);

void vec_mult()
{
   int i;
   init(v1, v2, N);
   #pragma omp target update to(v1, v2)
   #pragma omp target
   #pragma omp parallel for
   for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

   #pragma omp target update from(p)
   output(p, N);
}
```

- Indicate that global variables are mapped to a device data environment for the whole program
- Use `target update` to maintain consistency between host and device

`declare target` Construct

- Map variables to a device data environment for the whole program
- Syntax (C/C++)

```
#pragma omp declare target
    [variable-definitions-or-declarations]
#pragma omp end declare target
```

- Syntax (Fortran)

```
!$omp declare target (list)
```

- Clauses

```
to(extended-list)
link(list)
```

Call functions in a target region

- Function declaration must appear in a declare target construct
- The functions will be compiled for
  - → Host execution (as usual)
  - → Target execution (to be invoked from target regions)

```
#pragma omp declare target
float some_computation(float fl, int in) {
  // ... code ...
}

float final_computation(float fl, int in) {
  // ... code ...
}
#pragma omp end declare target
```

IWOMP 2016 Tutorial: OpenMP Accelerator Model

(OpenMP for Heterogeneous Computing)

Tom Scogland **Bronis R. de Supinski**

Lawrence Livermore National Laboratory

Review: Loop worksharing

Sequential code

```
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

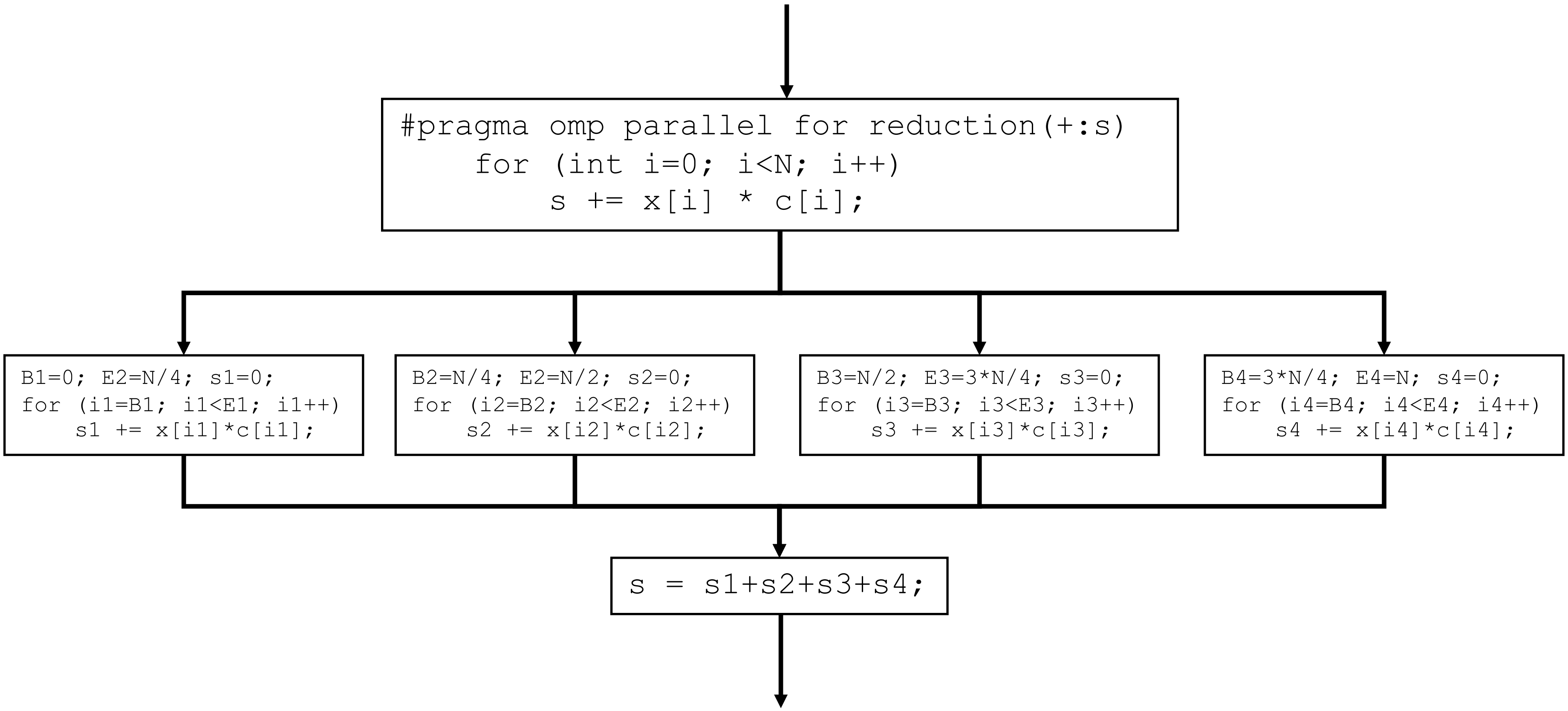
OpenMP Parallel Region

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

OpenMP Parallel Region and a work-sharing for construct

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

Review: Loop worksharing



- A single copy of x[] and c[] is shared by all threads

Terminology

- League:
  the set of threads teams created by a `teams` construct
- Contention group:
  threads of a team in a league and their descendant threads

`teams` Construct

- The **`teams`** construct creates a *league* of thread teams
  - The master thread of each team executes the **`teams`** region
  - The number of teams is specified by the **`num_teams`** clause
  - Each team executes with **`thread_limit`** threads
  - Threads in different teams cannot synchronize with each other

`teams` Construct – Restrictions

- A `teams` constructs must be “perfectly” nested in a `target` construct:
  - → No statements or directives outside the `teams` construct
- Only special OpenMP constructs can be nested inside a `teams` construct:
  - → `distribute` (see next slides)
  - → `parallel`
  - → `parallel for` (C/C++), `parallel do` (Fortran)
  - → `parallel sections`

`teams` Construct

- Syntax (C/C++):

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

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression)
thread_limit(integer-expression)
default(shared | none)
private(list), firstprivate(list)
shared(list), reduction(operator : list)
```

`distribute` Construct

- Worksharing construct for `target` and `teams` regions
  - → Distribute the iterations of the associated loops across the master threads of each team executing the region
  - → No implicit barrier at the end of the construct

- `dist_schedule(`*`kind[, chunk_size]`*`)`
  - → If specified scheduling kind must be static
  - → Chunks are distributed in round-robin fashion of chunks with size *`chunk_size`*
  - → If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

`distribute` Construct

- Syntax (C/C++):

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

- Syntax (Fortran):

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

- Clauses

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

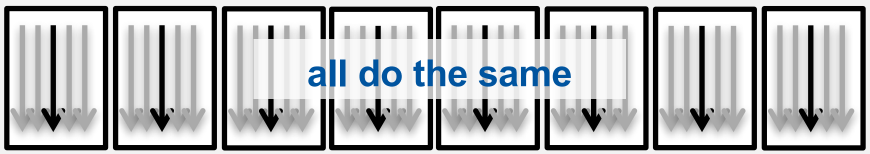
SAXPY: on device

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
#pragma omp target map(to:x[:n]) map(y[:n])
  {
#pragma omp parallel for
   for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
  }
}
```

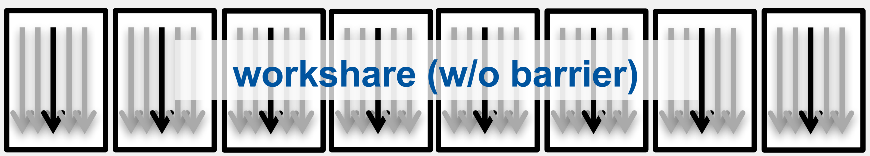
- How to run this loop in parallel on massively parallel hardware which typically has many clusters of execution units (or cores)?
- Chunk loop level 1: distribute big chunks of loop iterations to each cluster (thread blocks, coherency domains, card, etc...) – to each team
- Chunk loop level 2: loop workshare the iterations in a distributed chunk across threads in a team.
- Chunk loop level 3: Use SIMD-level parallelism inside each thread.

SAXPY: Accelerated worksharing

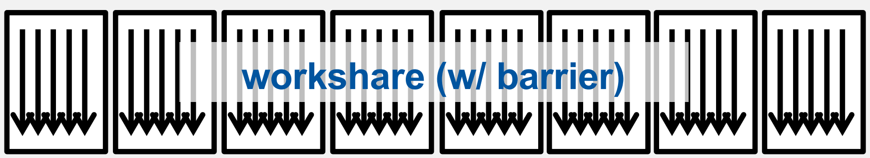
```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
#pragma omp target map(to:x[:n]) map(y[:n])
#pragma omp teams
{
 int block_size = n/omp_get_num_teams();
```




```
#pragma omp distribute dist_sched(static, 1)
  for (int i = 0; i < n; i += block_size){
```




```
#pragma omp parallel for [schedule(static:1)]
    for (int j = i; j < i + block_size; j++) {
```




```
        y[j] = a*x[j] + y[j];
}}
}}
```

Combined Constructs

- The distribution patterns can be cumbersome

- OpenMP defines composite constructs for typical code patterns
  - `distribute simd`
  - `distribute parallel for` (C/C++)
  - `distribute parallel for simd`(C/C++)
  - `distribute parallel do` (Fortran)
  - `distribute parallel do simd` (Fortran)
  - ... plus additional combinations for `teams` and `target`
- Avoids the need to do manual loop blocking

SAXPY: Combined Constructs

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{

#pragma omp target teams map(to:x[:n]) map(y[:n])
#pragma omp distribute parallel for
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

Comparing OpenMP with OpenACC

- **OpenMP 4.0 – accelerated workshare**

```
#pragma omp target teams map(B[0:N]) num_teams(numblocks)
#pragma omp distribute parallel for
for (i=0; i<N; ++1) {
  B[i] += sin(B[i]);
}
```

- **OpenACC – accelerated workshare**

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

IWOMP 2016 Tutorial: OpenMP Accelerator Model

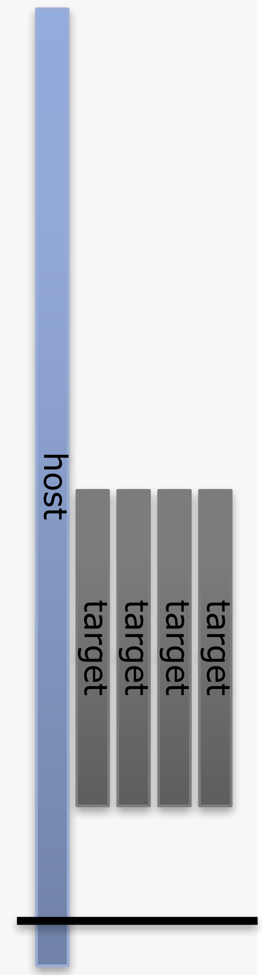
(OpenMP for Heterogeneous Computing)

Tom Scogland **Bronis R. de Supinski**

Lawrence Livermore National Laboratory

Multi-device Example

```
int num_dev = omp_get_num_devices() + 1;
int chunksz = length / num_dev;
assert((length % num_dev) == 0);
#pragma omp parallel sections firstprivate(chunksz,num_dev)
{
  for (int dev = 0; dev < num_dev; dev++) {
#pragma omp task firstprivate(dev)
    {
      int lb = dev * chunksz;
      int ub = (dev+1) * chunksz;
#pragma omp target if(num_dev > 1) device(dev) \
            map(in:y[lb:chunksz]) map(out:x[lb:chunksz])
      {
#pragma omp parallel for
        for (int i = lb; i < ub; i++) {
          x[i] = a * y[i];
        }
      }
    }
  }
}
```



`if` Clause Example (OpenMP 4.0)

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000

extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target if(N>THRESHOLD1) \
          map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for if(N>THRESHOLD2)
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

- The `if` clause on the `target` construct indicates that if the variable N is smaller than a given threshold, then the `target` region will be executed by the host device.
- The `if` clause on the `parallel` construct indicates that if the variable N is smaller than a second threshold then the parallel region is inactive.

`if` Clause Example (OpenMP 4.5)

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000

extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);

   #pragma omp target parallel for \
          if(target: N>THRESHOLD1) \
          if(parallel: N>THRESHOLD2) \
          map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

- OpenMP 4.5 added clause name modifiers for the `if` clause that allow the combined constructs to be used when you want to conditionally apply only part of the combined construct.

Array Sections Example

- If the type of the variable appearing in an array section is pointer, then the variable is implicitly treated as if it had appeared in a **map** clause as an array section of length zero with a *map-type* of **alloc**
- If any part of the original storage of a list item has corresponding storage in the enclosing device data environment, all of the original storage must have corresponding storage in the enclosing device data environment.

```
void foo (int *A, int N)
{
  int *p;

  #pragma omp target data map( A[:N])
  {
    // implicit firstprivate(A) for the pointer
    // A = storage allocated for array section on device
    p = &A[0];
    #pragma omp target map( p[0:N/2] )
    {
       A[N-1] = 0;
       p[0] = 0;
    }
  }
}
```

Tasks and target Example (OpenMP 4.0)

```
#include <stdlib.h>
#include <omp.h>

#pragma omp declare target
extern void init(float *, float *, int);
extern void output(float *, int);
extern void my_abort();
#pragma omp end declare target

void vec_mult(float *p, int N, int dev)
{
   float *v1, *v2;
   int i;
   init(p, N);

   #pragma omp task depend(out: v1, v2)
   #pragma omp target device(dev) map(v1, v2)
   {
      // check whether on device dev
      if (omp_is_initial_device())
         my_abort();
      v1 = malloc(N*sizeof(float));
      v2 = malloc(N*sizeof(float));
      init(v1,v2);
   }

   foo(); // execute asynchronously

   #pragma omp task depend(in: v1, v2)
   #pragma omp target device(dev) \
               map(to: v1, v2) \
               map(from: p[0:N])
   {
      // check whether on device dev
      if (omp_is_initial_device())
         my_abort();

      #pragma omp parallel for
      for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];

      output(p, N);
      free(v1);
      free(v2);
   }

   #pragma taskwait
} // end vec_mult()
```

Tasks and target Example (OpenMP 4.5)

```
#include <stdlib.h>
#include <omp.h>

#pragma omp declare target
extern void init(float *, float *, int);
extern void output(float *, int);
extern void my_abort();
#pragma omp end declare target

void vec_mult(float *p, int N, int dev)
{
   float *v1, *v2;
   int i;
   init(p, N);

   #pragma omp target nowait depend(out: v1, v2) \
               device(dev) map(v1, v2)
   {
      // check whether on device dev
      if (omp_is_initial_device())
         my_abort();
      v1 = malloc(N*sizeof(float));
      v2 = malloc(N*sizeof(float));
      init(v1,v2);
   }

   foo(); // execute asynchronously

   #pragma omp target nowait depend(in: v1, v2)\
               device(dev) \
               map(to: v1, v2) \
               map(from: p[0:N])
   {
      // check whether on device dev
      if (omp_is_initial_device())
         my_abort();

      #pragma omp parallel for
      for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];

      output(p, N);
      free(v1);
      free(v2);
   }

   #pragma taskwait
} // end vec_mult()
```