



OpenMP Advanced Training

SIMD, Tasks, and Teams

Doug Jacobsen, Intel Corporation
John Pennycook, Intel Corporation
Tim Mattson, Intel Corporation
Alejandro Duran, Intel Corporation

Intel, the Intel logo, Intel® Xeon Phi™, Intel® Xeon® Processor are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others. See [Trademarks on intel.com](https://www.intel.com/trademarks) for full list of Intel trademarks.



Topics

- We Will Cover:
 - Simplified OpenMP* basics review
 - Vectorization and OpenMP* SIMD
 - OpenMP* Tasking
 - OpenMP* Device Usage (target)
 - Brief Introduction to OpenMP* 5.0 Changes
- We will NOT Cover:
 - Detailed OpenMP* basics usage
 - Affinity
 - NUMA effects

Set up:

git clone [git@gitlab.lanl.gov:douglasj/openmp-training.git](https://gitlab.lanl.gov/douglasj/openmp-training.git)

git clone [git@github.com:douglasjacobsen/openmp_training.git](https://github.com/douglasjacobsen/openmp_training.git)

OpenMP* Review

OpenMP* Basics Review

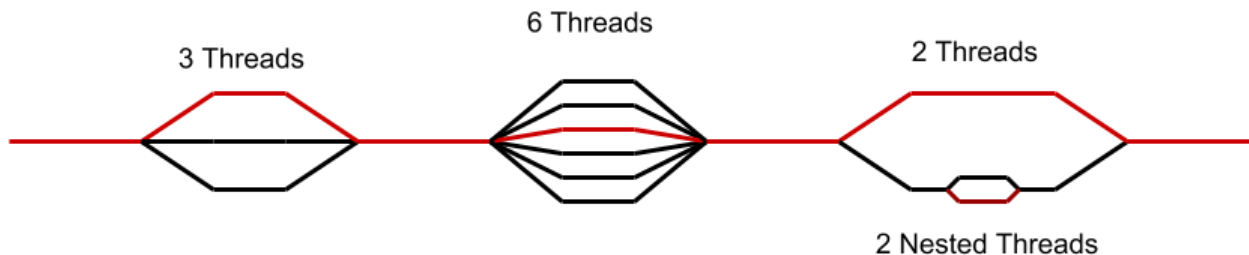
OpenMP*:

- Application Program Interface (API) focused on the expression of parallelism within an application
- Provides methods for creating multi-threaded, heterogeneous device, and vectorized applications
- Contains standardized directives and library functions
- Supports C, C++, and Fortran
- Was established in 1997

OpenMP* Threads

Threads are the original and most basic concept within OpenMP*.

A **master thread** spawns a team of threads when it encounters a parallel region



Threads can distribute work among themselves to accomplish the same task faster, or can work on independent tasks in parallel.

OpenMP* Directive Overview

Directive	Description
<code>#pragma omp parallel</code> <code>!\$omp parallel</code>	Create threads in a parallel region
<code>#pragma omp for</code> <code>!\$omp do</code>	Distribute iterations of a loop among threads
<code>#pragma omp for reduction(+:sum)</code> <code>!\$omp do reduction(+:sum)</code>	Distribute iterations of a loop among threads, and reduce the thread private 'sum' after the loop is complete.
<code>#pragma omp single</code> <code>!\$omp single</code>	Execute code with a single thread
<code>#pragma omp master</code> <code>!\$omp master</code>	Execute code only on the master thread
<code>#pragma omp barrier</code> <code>!\$omp barrier</code>	Wait for all threads in current team before proceeding

OpenMP* Data Sharing Attributes

Data Sharing Level	Description
shared	Variables are shared among all members. If one thread updates the value, all members see the new value.
private	Variables are private among members. If one member updates the value, no other member sees that update.
firstprivate	Same as private, but all members are initialized with the value before the scope changed.
lastprivate	Same as private, but the logically last iteration of a loop is used to set the value of the variable after the region ends.

OpenMP*: Beyond Threading

Modern computational architectures include everything from traditional CPUs, to Vector Processing Units, to Graphics Processing Units, and beyond.

OpenMP* is constantly evolving to help developers properly utilize the features of their underlying hardware.

The remainder of this talk will focus on more advanced topics from OpenMP*. These topics include:

- SIMD (Vectorization)
- Tasks
- Target Teams (Heterogeneous Devices)

Introduction to SIMD

Introduction to SIMD

Scalar Code

- Executes code one element at a time.

Vector Code

- Executes code multiple elements at a time.
- Single Instruction Multiple Data.

[Scalar] 1 elem at a time
`addss xmm1, xmm2`

[SSE] 4 elems at a time
`addps xmm1, xmm2`

[AVX] 8 elems at a time
`vaddps ymm1, ymm2, ymm3`

[MIC / AVX-512] 16 elems at a time
`vaddps zmm1, zmm2, zmm3`

Introduction to SIMD

- Not all scalar instructions have SIMD equivalents:
 - No support for certain data types in some ISAs.
 - Different widths for different data types.
 - Some instructions (e.g. jmp) make no sense in SIMD.
- Not all SIMD instructions have scalar equivalents:
 - “Find the max value in this register”
 - “Sum the values in this register”
 - “Load/store non-contiguous data”
 - “Rearrange the values in this register”

} “Horizontal”
operations

x86_64	SIMD
mov	vmov
add	vadd
mul	vmul
fmadd	vfmadd
rdtsc	-
jmp	-
clic	-
cmpxchg	-
cpuid	-
nop	-
-	vgather
-	vshuff
-	hadd

Introduction to SIMD

Hardware (Colors represent SIMD lanes)

3	1	2	4
---	---	---	---

 +

2	3	5	5
---	---	---	---

 =

5	4	7	9
---	---	---	---

Software (Vectorization)

```
for (int i = 0; i < size; i++) {  
    C[i] = A[i] + B[i];  
}
```

```
int i = 0;  
for (; i+3 < size; i += 4) {  
    for( int j = 0; j < 4; j++)  
        C[i+j] = A[i+j] + B[i+j];  
}  
for (; i < size; i++) {  
    C[i] = A[i] + B[i];  
}
```

Utilizing SIMD Instructions

Utilizing SIMD Instructions

SIMD Libraries

- Use a library that is already SIMD-optimized (e.g. Intel® Math Kernel Library)

Implicit (Auto) Vectorization

- Use a compiler that recognises vectorization opportunities (e.g. Intel® Composer XE)

Explicit (Manual) Vectorization

- Express vectorization opportunities to a compiler (e.g. OpenMP* 4.0)
- Write intrinsics code (or inline assembly!)

Utilizing SIMD Instructions – Auto-vectorization

Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;
void not_vectorizable
(float* a, float* b, float* c, int* ind)
{
    for (int i = 0; i < *g_size; i++)
    {
        int j = ind[i];
        c[j] += a[i] + b[i];
    }
}
```

- Unsafe Assumptions:
 - a, b and c point to different arrays.
 - Value of global g_size is constant.
 - ind[i] is a one-to-one mapping.

Utilizing SIMD Instructions – Auto-vectorization

Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;
void vectorizable
(float* restrict a, float* restrict b, float* restrict c, int* ind)
{
    int size = *g_size;
    #pragma ivdep
    for (int i = 0; i < size; i++)
    {
        int j = ind[i];
        c[j] += a[i] + b[i];
    }
}
```

▪ Safe Assumptions:

- a, b and c point to different arrays. (restrict)
- Value of global g_size is constant. (pointer dereference outside loop)
- ind[i] is a one-to-one mapping. (#pragma ivdep)

Utilizing SIMD Instructions – Compiler Pragmas

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```

- The same constructs can have different meaning from each other:
 - The two += operators have a different purpose.
 - The variables sum and p relate differently to the iteration space.
- The compiler has to generate different code.
- OpenMP* 4.0 pragmas allow programmers to express this.

Implicit vs Explicit Vectorization

Implicit

- Automatic dependency analysis.
(e.g. recognises SIMD reductions)
- Recognizes idioms with data dependencies.
(e.g. `array[i++] = x; -> vcompress`)
- Non-inline functions will be scalarized.
- Limited support for outer-loop vectorization (only with `-O3`).

Explicit

- No dependency analysis.
(e.g. SIMD reductions must be declared)
- Recognizes idioms without data dependencies.
- Non-inline functions can be vectorized.
- Outer loops can be vectorized.

OpenMP* SIMD

Explicit Vectorization – OpenMP* SIMD Loops

`#pragma omp simd / !$omp simd => for/do loop is a SIMD loop.`

Clause	Description
<code>private(list)</code>	Listed variables are scoped as private
<code>lastprivate(list)</code>	Listed variables are scoped as lastprivate
<code>reduction(reduction-identifier:list)</code>	Reduce listed variables according to reduction-identifier
<code>collapse(n)</code>	Associate loop construct with n following nested loops
<code>safelen(n)</code>	Maximum allowed 'distance' between SIMD lanes in a single execution
<code>simdlen(n)</code>	Preferred SIMD width
<code>aligned(list)</code>	Hint to the run-time to use aligned loads and stores on listed variables
<code>linear(list)</code>	Described later

Explicit Vectorization – OpenMP* 5.0 SIMD Loops

#pragma omp simd / !\$omp simd => for/do loop is a SIMD loop.

Clause	Description
nontemporal(list)	Hint to the runtime to use accesses with no temporal locality, i.e. there is no need to cache the data
order(concurrent)	Hint to the run-time that iterations of the loop can be executed in any order, including concurrently

Explicit Vectorization – OpenMP* SIMD Functions

`#pragma omp declare simd / !$omp declare simd => function called from a SIMD loop.`

Clause	Description
<code>simdlen(n)</code>	Preferred SIMD width is n
<code>aligned(list)</code>	Listed variables should use aligned accesses
<code>uniform(list)</code>	Listed variables have an invariant value for all concurrent calls to the SIMD function
<code>inbranch</code>	Function is always called from within a branch
<code>notinbranch</code>	Function is never called from within a branch
<code>linear(list)</code>	Described later

OpenMP* Linear Clause

The linear clause provides the compiler additional information about variables.
`linear(linear-list:step)`

Linear List Options	Description
variable val(variable)	Variable listed has linear values with a distance between consecutive SIMD lanes of step.
ref(variable)	Variable listed has linear memory addresses with a distance between consecutive SIMD lanes of step.
uval(variable)	Variable listed has linear values with a distance between consecutive SIMD lanes of step. Variable uses the same storage location across all SIMD lanes, and the logically last iteration is stored out.

OpenMP* SIMD Example

OpenMP* SIMD Example: STREAM Triad

Let's say we want to measure achievable bandwidth.

In order to achieve the highest bandwidth we can, we need to move a maximum number of bytes per cycle.

The STREAM Triad benchmark is typically used to measure this, it is a simple vector multiply and add of the form:

```
for ( int i = 0; i < N; i++ ) {  
    a[i] = b[i] * scalar + c[i];  
}
```

OpenMP* SIMD Exercise: STREAM Triad

Please perform these tasks for this exercise:

1. Checkout the **stream** branch
2. Build and run to determine what the achievable single core bandwidth of the base version is
``make``
``./STREAM.x 10000 10``
3. Add Vectorization to the main computational loop using OpenMP
(If you want to add nontemporal stores, let me know and I can help)
4. Build and run, to determine what your new achievable single core bandwidth is.
``make``
``./STREAM.x 10000 10``
5. Extra Credit: If you want to measure full socket bandwidth add OpenMP threading and re-run

You can find all examples at: [git@gitlab.lanl.gov:douglasj/openmp-training](https://gitlab.lanl.gov/douglasj/openmp-training)
or [git@github.com:douglasjacobsen/openmp_training](https://github.com/douglasjacobsen/openmp_training)

OpenMP* SIMD Example Answer: STREAM Triad

OpenMP 4.5

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i+=16) {
    #pragma vector nontemporal(a)
    #pragma omp simd aligned(a,b,c)
    for(j = 0; j < 16; j++) {
        a[i+j] = b[i+j] * scalar + c[i+j];
    }
}
```

OpenMP 5.0

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i+=16) {
    #pragma omp simd aligned(a,b,c) nontemporal(a)
    for(j = 0; j < 16; j++) {
        a[i+j] = b[i+j] * scalar + c[i+j];
    }
}
```

In this example, the 16 is tuned for single precision on AVX512

OpenMP* Task

OpenMP Tasks

Introduced in OpenMP* 3.0

An OpenMP* task consists of:

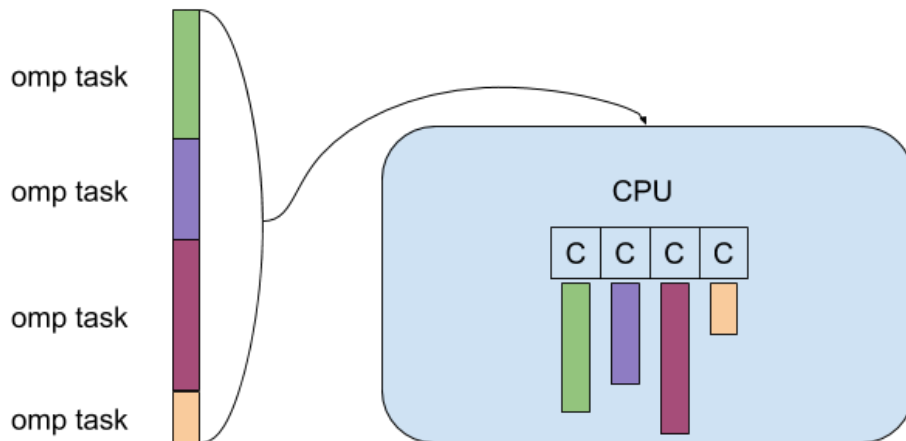
- Code to execute generated from a structured block
- Data environment generated from the data sharing clauses, and any defaults

Tasks can represent:

- Independent unrelated portions of work
- Subsequent iterations of a loop

OpenMP Tasks

When a thread (inside a parallel region) encounters a task construct a task is defined. The run-time may create a task, which threads can execute.



There are several different **types** of tasks, which we'll go over before introducing the task construct.

OpenMP* Task Types

Task Type	Description
Tied	Default task type. Once started, task cannot migrate to other threads.
Untied	Task can always migrate between threads.
Deferred	Default task type. May not run immediately when the task construct is encountered.
Undeferred	Task must complete before execution can continue. Does not require execution on the same thread.
Included	Creates an undeferred task that is executed immediately by the encountering thread.
Merged	The generated task uses the same data environment as the encountering thread.
Mergeable	A task that is able to be merged. Only if it is an undeferred or included task
Final	Generated task forces all of it's children to be final and included tasks
Predecessor / Dependent	Tasks that must be executed in order due to dependencies

OpenMP* Task Construct

`#pragma omp task / !$omp task =>` Generate a task from the following structured block

Clause	Description
<code>if([task:] scalar-expression)</code>	If scalar-expression evaluates to false, this is an undeferred task
<code>final(scalar-expression)</code>	If scalar-expression evaluates to true, this is a final task. Can reduce tasking overhead
<code>untied</code>	Generates untied tasks that can migrated between threads when preempted.
<code>default(private firstprivate shared none)</code>	Define the default data sharing value for this task
<code>mergeable</code>	Generates a mergeable task. Can reduce tasking overhead
<code>private(list)</code>	Variables listed will be scoped as private
<code>firstprivate(list)</code>	Variables listed will be scoped as firstprivate
<code>shared(list)</code>	Variables listed will be scoped as shared
<code>depend(dependence-type:list)</code>	Described later
<code>priority(N)</code>	Scalar value defining priority of tasks. Higher values imply tasks should be executed first. Does not guarantee an ordering.

OpenMP* 5.0 Task Construct Clauses

`#pragma omp task / !$omp task` => Generate a task from the following structured block

Clause	Description
<code>in_reduction(reduction-identifier:list)</code>	Generated task will participate in a previously defined <code>task_reduction</code> reduction. Listed variables are reduced according to <code>reduction-identifier</code> .
<code>allocate(allocator:list)</code>	Allocates private variables for generated task using <i>allocator</i>
<code>affinity(affinity-modifier:locator-list)</code>	Affinity hint that generated task should run close to listed storage locations
<code>detach(event-handle)</code>	Creates a new <i>allow-completion-event</i> event handle. Completion of the generated task is tied to the event.

OpenMP* Task Data Scoping

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: ??          -- Value: ??
            // Scope of b: ??          -- Value: ??
            // Scope of c: ??          -- Value: ??
            // Scope of d: ??          -- Value: ??
            // Scope of e: ??          -- Value: ??
        }
    }
}
```

OpenMP* Task Data Scoping

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared      -- Value: ??
            // Scope of b: firstprivate -- Value: ??
            // Scope of c: shared      -- Value: ??
            // Scope of d: firstprivate -- Value: ??
            // Scope of e: private     -- Value: ??
        }
    }
}
```

OpenMP* Task Data Scoping

```
int a = 1;
void foo() {
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task      Use default(none) to force explicit scoping.
        {
            int e = 5;
            // Scope of a: shared      -- Value: 1
            // Scope of b: firstprivate -- Value: Undefined! (Why?)
            // Scope of c: shared      -- Value: 3
            // Scope of d: firstprivate -- Value: 4
            // Scope of e: private     -- Value: 5
        }
    }
}
```

OpenMP* Task Dependencies

depend(dependence-type:list)

dependency-type options	Description
in	Generated task depends on any tasks with listed storage locations appearing in out or inout depend clauses
out	Allows other tasks to depend on execution if listed storage locations appear in their in or inout clauses
inout	Combines the two in and out dependence types
mutexinoutset (in 5.0)	Generates a mutually exclusive task with other tasks that have the same listed storage locations in a mutexinoutset. Creates dependencies the same way inout does.

NOTE: Can use iterators to build dependencies in 5.0

OpenMP* Taskloop Construct

`#pragma omp taskloop / !$omp taskloop` => Generate tasks from subsets of the iteration space.

Clause	Description
<code>if([taskloop:] scalar-logical-expression)</code>	If scalar-logical-expression evaluates to false, undeferred tasks are generated.
<code>shared(list)</code>	Listed variables are scoped as shared
<code>private(list)</code>	Listed variables are scoped as private
<code>firstprivate(list)</code>	Listed variables are scoped as firstprivate
<code>lastprivate(list)</code>	Listed variables are scoped as lastprivate
<code>default(private firstprivate shared none)</code>	Sets default variable scoping
<code>grainsize(grain-size)</code>	Specifies the minimum number of iterations bundled into a task
<code>num_tasks(num-tasks)</code>	Specifies the number of tasks to create
<code>collapse(n)</code>	Associates this taskloop directive with the following n nested loops
<code>final(scalar-logical-expression)</code>	If scalar-logical-expression evaluates to true, all tasks generated are final and included.
<code>priority(n)</code>	Hint to the run-time that higher priority tasks should execute before lower priority tasks.

OpenMP* Taskloop Construct cont.

#pragma omp taskloop / !\$omp taskloop => Generate tasks from subsets of the iteration space.

Clause	Description
untied	All generated tasks are untied tasks
mergeable	All generated tasks are mergeable
nogroup	No implicit taskgroup construct is created for generated tasks
reduction([default,] reduction-identifier : list) (in 5.0)	Creates tasks as if task_reduction clause was on each task
in_reduction(reduction-identifier: list) (in 5.0)	Creates tasks as if in_reduction clause was on each task
allocate([allocator:]list) (in 5.0)	Allocates listed variables using allocator

```
void foo(int n, int *v){
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop
    for ( int i = 0; i < n; ++i)
    {
        compute(v[i]);
    }
}
```

OpenMP* Taskgroup Construct

`#pragma omp taskgroup / !$omp taskgroup` => Groups tasks and their children into an execution environment.

There is an implicit synchronization at the end of a taskgroup construct.

Clause	Description
<code>task_reduction([default,] reduction-identifier : list)</code> (in 5.0)	Defines a reduction among tasks. Tasks participating in the reduction need an <code>in_reduction</code> clause.
<code>allocate([allocator:]list)</code> (in 5.0)	Allocates listed variables using allocator

```
void tg(){
    #pragma omp parallel
    #pragma omp taskgroup
    {
        #pragma omp task
        compute_a();
        #pragma omp task
        compute_b();
        #pragma omp task
        compute_c();
    }
}
```


OpenMP* Additional Task Directives

In addition to **#pragma omp task** OpenMP* also includes support for:

- **#pragma omp taskloop simd / !\$omp taskloop simd**
 - Generates tasks that each execute iterations from one or more loops in a SIMD fashion
- **#pragma omp taskyield / !\$omp taskyield**
 - Specifies that the current task can be suspended in favor of a different task
- **#pragma omp taskwait / !\$omp taskwait**
 - Specifies a wait on the completion of all child tasks of the current task
- **#pragma omp cancel / !\$omp cancel**
 - Allows cancellation of a construct, from within the construct.
Available constructs are {parallel, sections, for / do, taskgroup}
Needs OMP_CANCELLATION=true for cancel to work
- **#pragma omp cancellation point / !\$omp cancellation point**
 - User defined point where members of a construct check if it has been canceled

OpenMP* Tasking

Two common ways which tasks are used are:

- Single Creator:
 - Threads are spawned
 - Most threads wait for a task, while a single thread creates all of the tasks
 - As tasks are created, the waiting threads start executing them in parallel
- Multiple Creators:
 - Threads are spawned
 - A worksharing construct is used to distribute iterations among threads
 - Each iteration (or some number of iterations) create a task

OpenMP* Task Example: Fibonacci

We are going to play with the Fibonacci series to demonstrate tasking.

Fibonacci Series: 0 1 1 2 3 5 8 13 21 ...

$$i_n = i_{n-1} + i_{n-2}$$

If $n = 0$ or 1 , $i_n = n$

```
int fibonacci(int n){  
    int x, y;  
    if ( n < 2 ) return n;  
  
    x = fibonacci(n-1);  
    y = fibonacci(n-2);  
  
    return (x + y);  
}
```

OpenMP* Task Exercise: Fibonacci

Fibonacci Exercises:

1. Checkout the **fibonacci** branch
2. Build and run to see how fast you can compute the 40th Fibonacci number
./FIB.x 40
3. Add OpenMP tasking to compute x and y in parallel
4. Build and run with the same number and compare to your previous time
5. Extra Credit: Add the **final** and **mergeable** clauses to prevent tasks at lower levels of the tree (i.e. $n = 10$)
Benchmark again
6. Extra Credit: Compare some other Fibonacci number times

You can find all examples at [git@gitlab.lanl.gov:douglasj/openmp-training](https://gitlab.lanl.gov/douglasj/openmp-training)

OpenMP* Task Example: Fibonacci Answer

```
int fibonacci(int n){
    int x, y;
    if ( n < 2 ) return n;

    #pragma omp task shared(x) mergeable final(n < 10)
    x = fibonacci(n-1);
    #pragma omp task shared(y) mergeable final(n < 10)
    y = fibonacci(n-2);
    #pragma omp taskwait

    return(x + y);
}

int main(int argc, char **argv){
    ....
    #pragma omp parallel
    {
        #pragma omp master
        {
            fibonacci(n);
        }
    }
    ....
}
```

OpenMP* Task Example: Task based search (and cancellation)

In this exercise, we are searching for a value among unsorted bins.

We will use tasks to parallelize the bin search, and attempt to add an early exit using the cancel directive.

```
void search( const int search_val) {  
    for ( int i = 0; i < table.size(); i++ ) {  
        for( int j = 0; j < table[i].size(); j++ ) {  
            if ( table[i][j] == search_val ) {  
                printf("Value found in bin %d at index %d\n",  
                    i, j);  
            }  
        }  
        searched[i] = 1;  
    }  
}
```

OpenMP* Task Example: Task based search (and cancellation)

Search exercises

1. Checkout the **search** branch
2. Build and run with an example of
`./SRCH.x 1000000 300 15`
3. Add tasking (and a task group) to the search loop within the `search_table` function
4. Add a cancel and cancellation point to exit the task group after the value has been found.

You can find all examples at [git@gitlab.lanl.gov:douglasj/openmp-training](https://gitlab.lanl.gov/douglasj/openmp-training)

OpenMP* Task Example: Task based search (and cancellation)

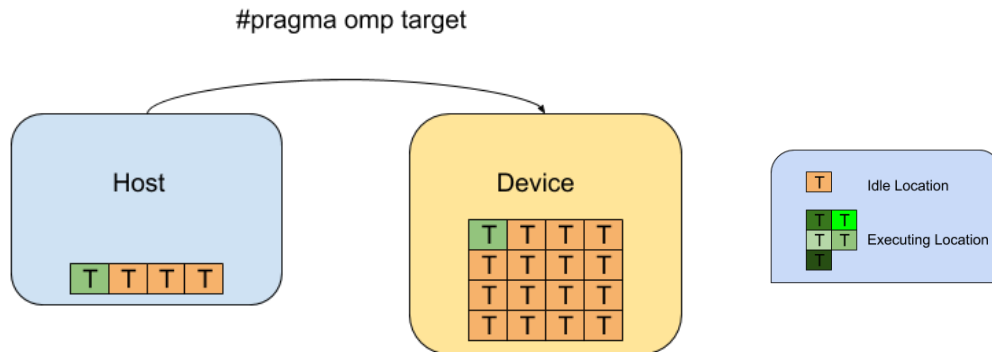
```
void search( const int search_val) {  
  
    #pragma omp parallel  
    {  
        #pragma omp master  
        {  
            #pragma omp taskgroup  
            for ( int i = 0; i < table.size(); i++ ) {  
                #pragma omp task  
                {  
                    for( int j = 0; j < table[i].size(); j++ ) {  
                        #pragma omp cancellation point taskgroup  
                        if ( table[i][j] == search_val ) {  
                            printf("Value found in bin %d at index %d\n",  
                                i, j);  
                        }  
                        #pragma omp cancel taskgroup  
                    }  
                    searched[i] = 1;  
                }  
            }  
        }  
    }  
}
```


OpenMP* Target

OpenMP* Target

Introduced in OpenMP* 4.0

The **target** construct is used to create a *target task* that should be offloaded to a device.



We'll carry the STREAM triad example with us to show device execution. Follow along, starting with the **stream_target_start** branch

You can find all examples at git@gitlab.lanl.gov:douglasj/openmp-training
or git@github.com:douglasjacobsen/openmp_training

OpenMP* Target Construct

`#pragma omp target / !$omp target`
`!$omp end target` => Create a *target task* task, and execute on a device

Clause	Description
<code>if(scalar-expression)</code>	If scalar-expression is false, device used is host
<code>device(integer-expression)</code>	Controls the device used, as described in data management constructs
<code>private(list)</code>	Listed variables are scoped as private
<code>firstprivate(list)</code>	Listed variables are scoped as firstprivate
<code>map([[map-type-modifier[,]]map-type:]list)</code>	Described later
<code>is_device_ptr(list)</code>	Listed variables are already device pointers (allocated before the target construct)
<code>defaultmap(tofrom:scalar)</code>	If specified, scalars are by default mapped using tofrom. If not specified, nothing is mapped by default.
<code>depend(dependence-type:list)</code>	Follows task dependencies, as previously described.
<code>nowait</code>	If present, execution of the target task may be deferred. If not, the task is an undeferred task.

OpenMP* 5.0 Target Construct

`#pragma omp target / !$omp target`
`!$omp end target` => Create a *target task* task, and execute on a device

Clause	Description
<code>in_reduction(reduction-identifier:list)</code>	The generated target task will participate in a previously defined <code>task_reduction</code>
<code>defaultmap(implicit-behavior:[variable-category])</code>	Implicit behavior can be one of <code>alloc</code> , <code>to</code> , <code>from</code> , <code>tofrom</code> , <code>firstprivate</code> , <code>none</code> , or <code>default</code> and variable category can be one of <code>scalar</code> , <code>aggregate</code> , <code>pointer</code> , or <code>allocatable</code> (in fortran)
<code>allocate([allocator:]list)</code>	Allocates listed variables using the specified allocator (or a default device allocator)
<code>uses_allocators(allocator[(allocator-traits-arrays)], ...)</code>	Each allocator listed will be made available in the target region.
<code>device([device-modifier:]integer-expression)</code>	Modifier can be <code>ancestor</code> , or <code>device_num</code> . Used to determine which device will be selected. <code>ancestor:1</code> executes on the parent device of the target region.

OpenMP* Target Data Mapping

`map([[map-type-modifier[,]]map-type:]list)`

Clause	Valid for construct	Description
to	target enter data, target data, target	Copies data from listed variables into device data region
from	target exit data, target data, target	Copies data from listed variables out of the device data region
tofrom	target data, target	Copies data into and out of the device data region for listed variables
alloc	target enter data, target data, target	Allocates memory for listed variables in the device data region
release	target exit data	Releases device pointers for listed variables in the device data region
delete	target exit data	Frees memory for listed variables in the device data region

Structured data regions trigger events upon entry and exit.

In 5.0 `requires(unified_shared_memory)` can make `map()` optional

OpenMP* Target Example: STREAM Triad

Always Move Everything

```
#pragma omp target map(tofrom:a[0:N], b[0:N], c[0:N], scalar)
for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

Only Move What's Needed

```
#pragma omp target map(to:b[0:N], c[0:N], scalar) map(tofrom:a[0:N])
for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

These examples are in the **stream_target_1** branch

OpenMP* Target Structured Data Management

`#pragma omp target data / !$omp target data`
`!$omp end target data` => Create a data region for a device

Clause	Description
<code>if(scalar-expression)</code>	If scalar-expression is false, device used is host
<code>device(integer-expression)</code>	Integer expression for the device to use. Must be a non-negative integer less than the value of <code>omp_get_num_devices()</code>
<code>map([[map-type-modifier[,]]map-type:]list)</code>	Described previously
<code>use_device_ptr(list)</code>	Listed items are converted to device pointers in the device data environment
<code>use_device_addr(list)</code> (in 5.0)	Listed items have the address of the corresponding object in the device data environment.

OpenMP* Target Unstructured Data Management

#pragma omp enter target data / !\$omp enter target data

#pragma omp exit target data / !\$omp exit target data => Manage data associated with a device

Clause	Description
if(scalar-expression)	If scalar-expression is false, device used is host
device(integer-expression)	Integer expression for the device to use. Must be a non-negative integer less than the value of <code>omp_get_num_devices()</code> If not specified, device is determined from the default-device-var icv.
map([[map-type-modifier[,]]map-type:]list)	Described previously
depend(dependence-type:list)	Follows task dependencies, as previously described.
nowait	If present, execution of the target task may be deferred. If not, the task is an undeferred task.

OpenMP* Target Example: STREAM Triad

Structured Data Management

```
#pragma omp target data map(to:b[0:N],c[0:N],scalar) \
                        map(from:a[0:N])
{
  #pragma omp target
  for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
  }
}
```

Unstructured Data Management

```
#pragma omp target enter data map(to:a[0:N], b[0:N], \
                                c[0:N],scalar)

#pragma omp target
for (i = 0, i < N, i++){
  a[i] = b[i] * scalar + c[i];
}

#pragma omp target exit data map(from:a[0:N])
```

These examples are in the **stream_target_2** branch

OpenMP* Target Example: STREAM Triad

```
#pragma omp target enter data \  
    map(to:a[0:N],b[0:N],c[0:N])  
  
#pragma omp target  
for (i = 0, i < N, i++){  
    a[i] = b[i] * scalar + c[i];  
}  
  
#pragma omp target exit data map(from:a[0:N])
```

At the beginning of the target region, b, c, and scalar will be copied onto the device and after the target region is complete, a will be copied back to the host. The target region will be executed on the device.

However....

All of the examples so far have a **serial** execution!

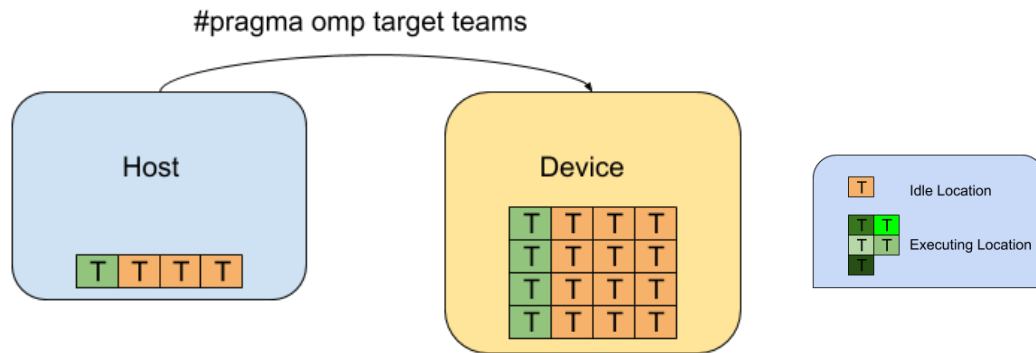
The target construct only creates a target task which is executed by a **single thread**!

OpenMP* Teams and Distribute

OpenMP* Teams

To utilize multiple threads on a device, we need to first use the **teams** construct.

A teams construct creates a league of teams. Each team consists of some number of threads, and to begin with the **master** thread of each team executes the code in the teams region.



OpenMP* Teams Construct

```
#pragma omp teams/ !$omp teams  
!$omp end teams =>
```

Create a league of thread teams. The master thread of each team executes the region.

Clause	Description
num_teams(integer-expression)	Sets the maximum number of teams to create within a target task
thread_limit(integer_expression)	Sets the maximum number of threads to create within each team
default(shared firstprivate private none)	Sets the default data sharing level
private(list)	Listed variables are scoped as private
firstprivate(list)	Listed variables are scoped as firstprivate
shared(list)	Listed variables are scoped as shared
reduction(reduction-identifier:list)	Perform a reduction on listed variables across teams using reduction-identifier.

OpenMP* Teams Example: STREAM Triad

```
#pragma target teams  
  
for (i = 0, i < N, i++){  
    a[i] = b[i] * scalar + c[i];  
}
```

Now we have multiple threads executing on the device, however they are all performing the **same** work.

Additionally, only the **master** thread of each team is doing anything.

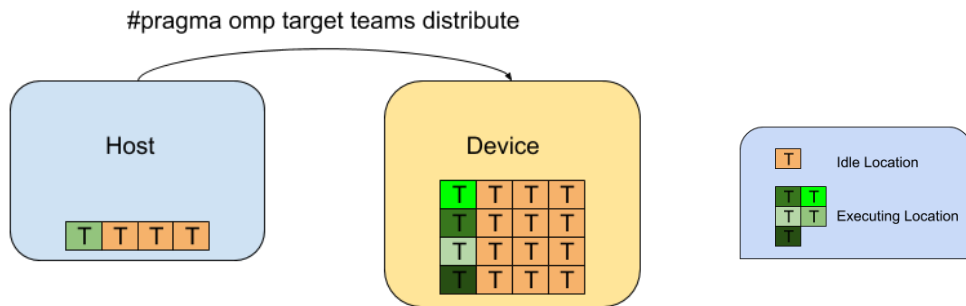
This example is in the **stream_target_3** branch

OpenMP* Distribute

Right now we have:

- Multiple thread teams working on a device
- All thread teams performing the same work
- Only the **master** thread of each team executing anything

When we encounter a loop, we can spread the iterations among teams using the **distribute** construct.



OpenMP* Distribute Construct

`#pragma omp distribute / !$omp distribute
 !$omp end distribute` =>

Distribute the iterations of one or more loops across the master threads of a league of teams

Clause	Description
<code>private(list)</code>	Listed variables are scoped as private
<code>firstprivate(list)</code>	Listed variables are scoped as firstprivate
<code>lastprivate(list)</code>	Listed variables are scoped as lastprivate
<code>collapse(n)</code>	Specifies how many loops are associated with a particular distribute construct
<code>dist_schedule(kind[, chunk_size])</code>	Kind must be <i>static</i> , <i>chunk_size</i> specifies the division of iterations. Chunks are assigned to teams in a round-robin fashion

OpenMP* Distribute Example: STREAM Triad

```
#pragma target teams distribute  
  
for (i = 0, i < N, i++){  
    a[i] = b[i] * scalar + c[i];  
}
```

At this point, we have teams performing independent iterations of this loop, however each team is only executing serially.

In order to work in parallel **within a team** we thread as we normally would with the basic OpenMP usage.

However, there are composite constructs that can help.

This example is in the **stream_target_4** branch

OpenMP* Distribute Composite Constructs

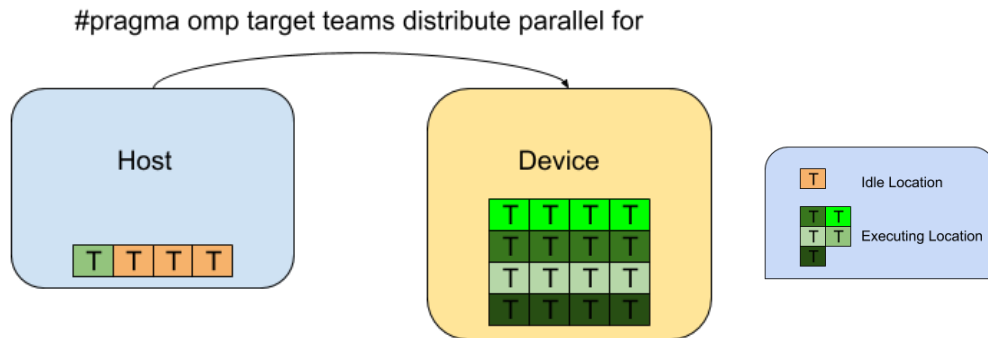
In addition to using a vanilla *distribute* construct, there are also composite constructs that allow more parallelism.

Construct	Description
<code>#pragma omp distribute simd</code> <code>!\$omp [end] distribute simd</code>	Distribute loop chunks among teams, and execute those iterations using SIMD instructions
<code>#pragma omp distribute parallel for</code> <code>!\$omp [end] distribute parallel do</code>	Distribute loop chunks among teams, then distribute iterations within each chunk among threads within the team and execute in parallel
<code>#pragma omp distribute parallel for simd</code> <code>!\$omp [end] distribute parallel for simd</code>	Distribute loop chunks among teams, then distribute iterations within each chunk among threads within the team and execute in parallel using SIMD instructions

OpenMP* Distribute Full Example: STREAM Triad

```
#pragma target teams distribute parallel for  
  
for (i = 0, i < N, i++){  
    a[i] = b[i] * scalar + c[i];  
}
```

Now, we have teams splitting the work into large chunks on the device, while threads within the team will further split the work into smaller chunks and execute in parallel.



This example is in the **stream_target_5** branch

OpenMP* 5.0

OpenMP* 5.0 adds several new features which can be useful in achieving a performance portable application. We won't go over these in detail, but some items to look into are:

Construct	Description
<code>#pragma omp metadirective</code> <code>!\$omp metadirective</code>	Conditionally apply a directive from a set to a location.
<code>#pragma omp declare variant</code> <code>!\$omp declare variant</code>	Define variants of a function to use on different architectures or devices.
<code>#pragma omp requires</code> <code>!\$omp requires</code>	Specify that an implementation requires a specific hardware feature
<code>#pragma omp teams loop</code> <code>!\$omp teams loop</code>	Specify a teams construct that contains a loop and nothing else Equivalent to <code>#pragma omp teams distribute parallel for</code> <code>!\$omp teams distribute parallel for</code>
<code>#pragma omp scan</code> <code>!\$omp scan</code>	Tells the compiler that the following statement is a scan that can be performed in parallel
Memory Management <code>#pragma omp allocate</code> <code>!\$omp allocate</code>	More explicit control over memory spaces for host and device, and the ability to define and use custom memory allocators

Example Answers

All example answers can be found at:

[git@gitlab.lanl.gov:douglasj/openmp-training-answers](https://gitlab.lanl.gov/douglasj/openmp-training-answers)

[git@github.com:douglasjacobsen/openmp_training_answers](https://github.com/douglasjacobsen/openmp_training_answers)

For additional resources, see:

<http://www.nersc.gov/assets/Uploads/OMP-beyond-the-common-core-20150504-final.pdf>

<https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

<https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>

OpenMP* Target (Optional) Extra Example: STREAM Triad

1. Starting from the **stream** branch, add unstructured data management to the arrays, to map them onto the target device immediately after they are allocated.
2. Offload the array initialization
3. Offload the triad computation
4. Use unstructured data management to pull the result array back to the host
5. Add a test to ensure the device computed the correct answer
6. Build / Run

Legal Disclaimers

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. § For more information go to www.intel.com/benchmarks.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel, the Intel logo, Look Inside, Xeon, Xeon Phi, VTune, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© Intel Corporation.

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 SSE3 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

