Advanced OpenMP Tutorial

OpenMPCon & IWOMP 2017

Christian Terboven
Michael Klemm

Members of the OpenMP Language Committee

IWOMP 2017

Credits

The slides are jointly developed by:

Christian Terboven — RWTH AACHEN UNIVERSITY

Michael Klemm — intel

Ruud van der Pas — ORACLE

Eric Stotzer — TEXAS INSTRUMENTS

Bronis R. de Supinski — LAWRENCE LIVERMORE NATIONAL LABORATORY

Agenda

Topic	Speaker	Time
OpenMP Tasking	Christian	45 min
Vectorization	Michael	45 min
Short Break	All ☺	15 min
NUMA Awareness	Christian	30 min
Offload Programming	Michael	45 min

OpenMP Tasking

Agenda

- Intro by Example: Sudoku
- Data Scoping
- Scheduling and Dependencies
- Taskloops
- More Tasking Stuff

Intro by Example: Sudoku

Sudoko for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

(1) Search an empty field

(2) Try all numbers:

(2 a) Check Sudoku

(2 b) If invalid:
skip

(2 c) If valid:
Go to next field

Wait for completion

The OpenMP Task Construct

C/C++

```
#pragma omp task [clause]
... structured block ...
```

Fortran

```
!$omp task [clause]
... structured block ...
!$omp end task
```

- Each encountering thread/task creates a new task
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another task directive
    - → Into a Worksharing construct
- Data scoping clauses:
  - → `shared`(*list*)
  - → `private`(*list*) `firstprivate`(*list*)
  - → `default`(*shared* | *none*)

Barrier and Taskwait Constructs

- OpenMP `barrier` (implicit or explicit)
  - → All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

```
C/C++
#pragma omp barrier
```

- Task barrier: `taskwait`
  - → Encountering task is suspended until child tasks complete
    - → Applies only to children, not descendants!

```
C/C++
#pragma omp taskwait
```

Parallel Brute-force Sudoku

- This parallel algorithm finds all valid solutions

	6														
15	11														
13		9	12												
2		16		11											
	15	11	10												
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9											
16	14			7											
11	10		15												
		12		1	4	6		16				11	10		
		5		8											
3	16			10											

first call contained in a
`#pragma omp parallel`
`#pragma omp single`
such that one tasks starts the execution of the algorithm

`#pragma omp task`
needs to work on a new copy of the Sudoku board

`#pragma omp taskwait`
wait for all child tasks

(1) Search an empty field

(2) Try all numbers:

(2 a) Check Sudoku

(2 b) If invalid:
skip

(2 c) If valid:
Go to next field

Wait for completion

Parallel Brute-force Sudoku (2/3)

- OpenMP parallel region creates a team of threads

```
#pragma omp parallel
{
#pragma omp single
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

  → Single construct: One thread enters the execution of `solve_parallel`

  → the other threads wait at the end of the `single` …

    → … and are ready to pick up threads „from the work queue“

- Syntactic sugar (either you like it or you don‘t)

```
#pragma omp parallel sections
{
   solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

Parallel Brute-force Sudoku (3/3)

- The actual implementation

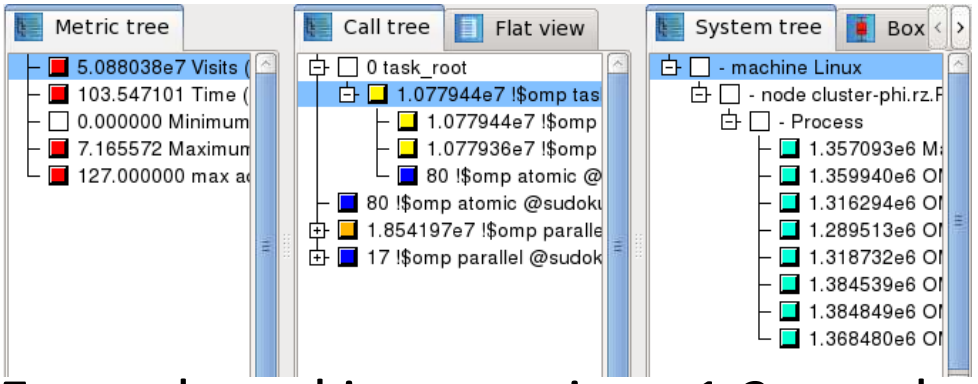
```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
{
      // create from copy constructor
      CSudokuBoard new_sudoku(*sudoku);
      new_sudoku.set(y, x, i);
      if (solve_parallel(x+1, y, &new_sudoku)) {
         new_sudoku.printBoard();
      }
} // end omp task
   }
}

#pragma omp taskwait
```

`#pragma omp task`
need to work on a new copy of the Sudoku board

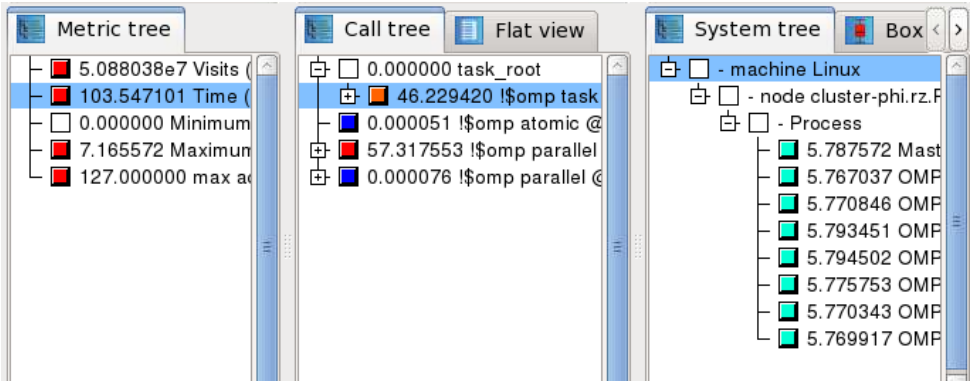
`#pragma omp taskwait`
wait for all child tasks

Performance Analysis

Event-based profiling gives a good overview :



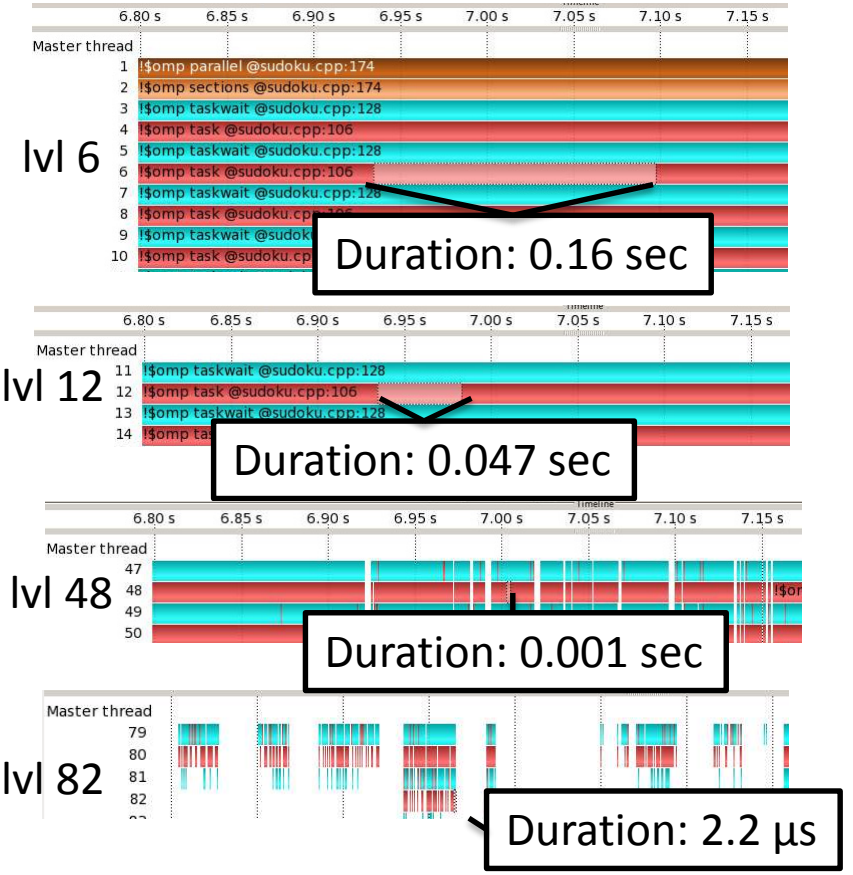
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

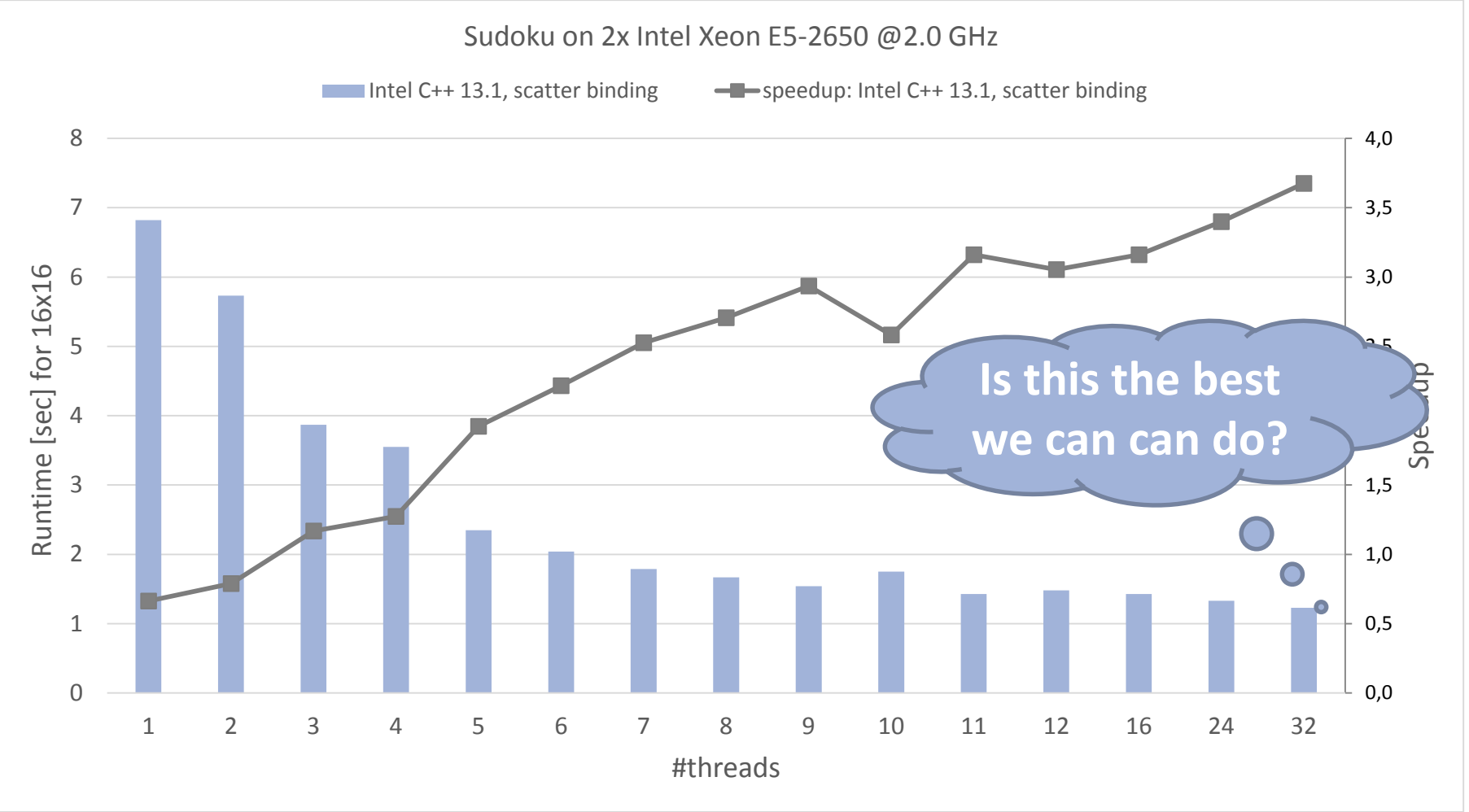
=> average duration of a task is ~4.4 µs

Tracing gives more details:



Tasks get much smaller down the call-stack.

Performance Evaluation



Performance Analysis

Event-based profiling gives a good overview :

Tracing gives more details:

Performance and Scalability Tuning Idea: If you have created sufficiently many tasks to make you cores busy, stop creating more tasks!

- if-clause
- final-clause, mergeable-clause
- natively in your program code

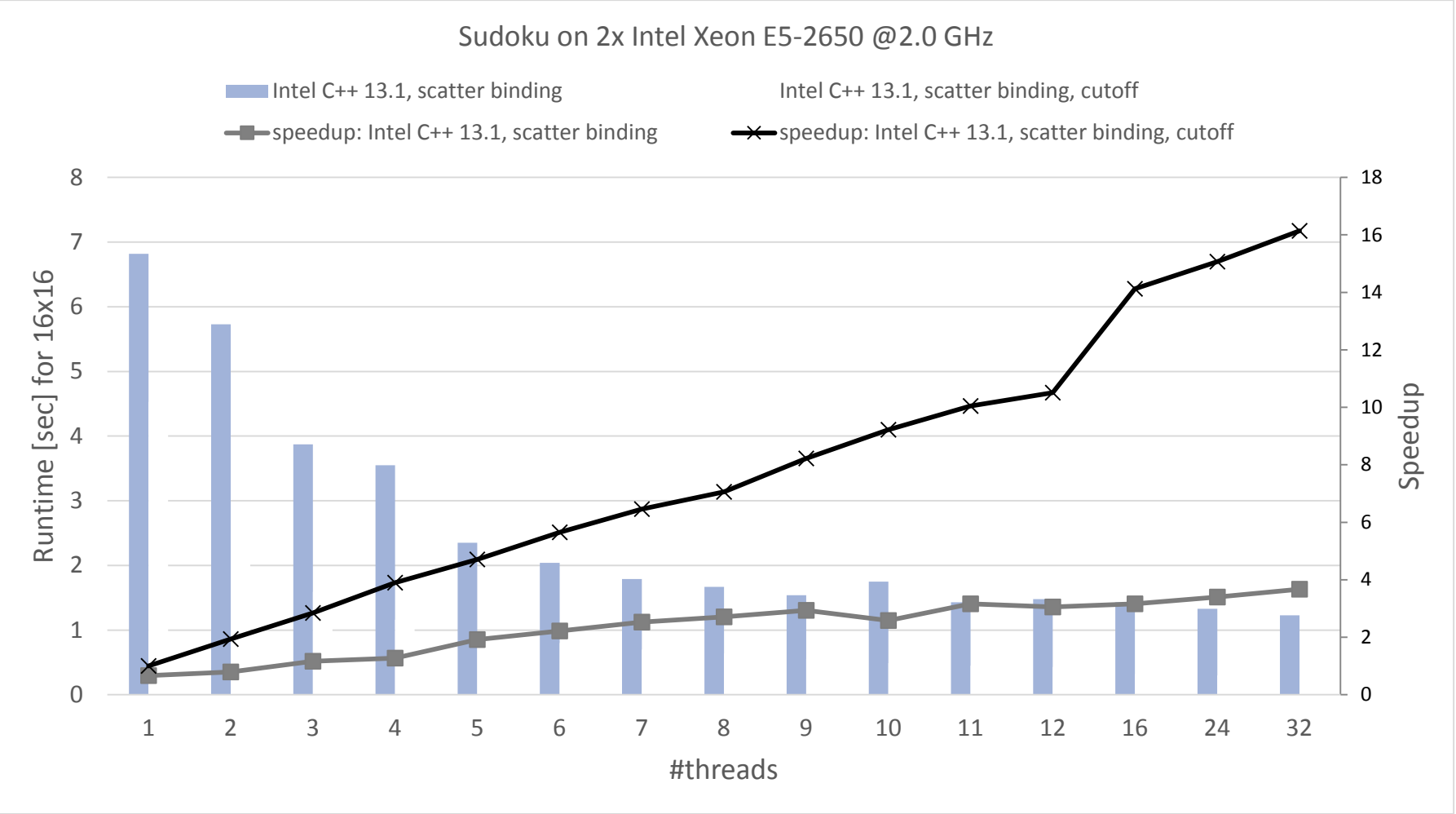
Example: stop recursion

… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

Tasks get much smaller down the call-stack.

Performance Evaluation



Data Scoping

Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply:
  - → Static and Global variables are shared
  - → Automatic Storage (local) variables are private

- If `shared` scoping is not inherited:
  - → Orphaned Task variables are `firstprivate` by default!
  - → Non-Orphaned Task variables inherit the `shared` attribute!
  - → Variables are `firstprivate` unless `shared` in the enclosing context

Data Scoping Example (1/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a:
                // Scope of b:
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (2/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b:
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (3/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (4/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (5/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e:
} } }
```

Data Scoping Example (6/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```

Data Scoping Example (7/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared,        value of a: 1
                // Scope of b: firstprivate,  value of b: 0 / undefined
                // Scope of c: shared,        value of c: 3
                // Scope of d: firstprivate,  value of d: 4
                // Scope of e: private,       value of e: 5
} } }
```

Use default(none)!

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b) default(none)
   #pragma omp parallel private(b) default(none)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```

Scheduling and 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 task scheduling points
    - → 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
  - → Resume at task scheduling points possibly by different thread
  - → ~~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` variable
  - → Avoid any use of thread-ids (i.e., `omp_get_thread_num()`)
  - → Be careful with `critical region` and *locks*

- Possible solution:
  - → Create a tied task region with

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

`if` Clause

- 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 the new task finishes
  - → Used for optimization, e.g., avoid creation of small tasks

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++

```
#pragma omp taskyield
```

Fortran

```
!$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; may also avoid deadlock situations.

priority Clause

```
C/C++
#pragma omp task priority(priority-value)
... structured block ...
```

- The *priority* is a hint to the runtime system for task execution order
- Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones
  - → priority is non-negative numerical scalar (default: 0)
  - → priority <= max-task-priority ICV
    - → environment variable OMP_MAX_TASK_PRIORITY
- It is not allowed to rely on task execution order being determined by this clause!

`final` Clause

- For recursive problems that perform task decomposition, stopping task creation at a certain depth exposes enough parallelism but reduces overhead.

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

- 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 might merge the task's data environment
  - → 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, no compiler or runtime exploits final and/or mergeable so that real world applications would profit from using them ☹.

The taskgroup Construct

C/C++

```
#pragma omp taskgroup
... structured block ...
```

Fortran

```
!$omp taskgroup
... structured block ...
!$omp end task
```

- Specifies a wait on completion of child tasks and their descendant tasks
  - → „deeper“ sychronization than `taskwait`, but
  - → with the option to restrict to a subset of all tasks (as opposed to a `barrier`)

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.

Degree of parallism exploitable in this concrete example:
T2 and T3 (2 tasks), T1 of next iteration has to wait for them

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

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

T2 and T3 can be executed in parallel.

```
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
}
```

Concurrent Execution w/ Dep.

- The following code allows for more parallelism, as there is one i per thread. Thus, two tasks may be active per thread.

```
void process_in_parallel() {
   #pragma omp parallel
   {
      #pragma omp for
      for (int i = 0; i < T; ++i) {
         #pragma omp task depend(out: i)
            preprocess_some_data(...);
         #pragma omp task depend(in: i)
            do_something_with_data(...);
         #pragma omp task depend(in: i)
            do_something_independent_with_data(...);
      }
   } // end omp parallel
}
```

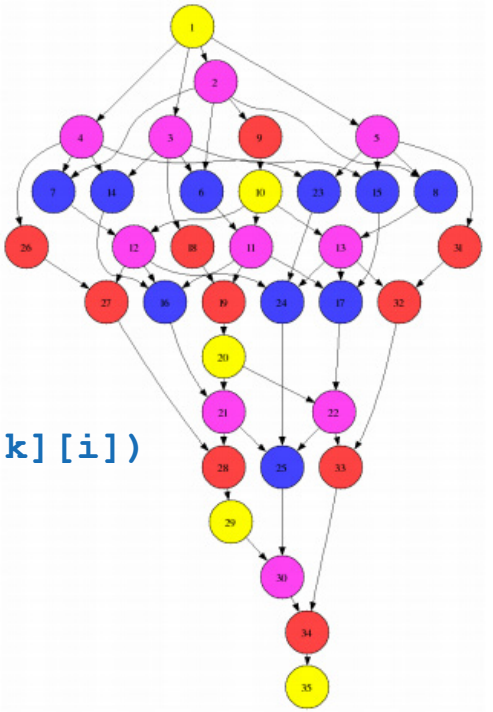
Concurrent Execution w/ Dep.

- The following allows for even more parallelism, as there now can be two tasks active per thread per i-th iteration.

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



* image from BSC

„Real“ Task Dependencies

Jack Dongarra on OpenMP Task Dependencies:

[…] The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. Until now, libraries like PLASMA had to rely on custom built task schedulers; […] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory. **We view OpenMP as the natural path forward for the PLASMA library and expect that others will see the same advantages to choosing this alternative.**

Full article here: http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/

* image from BSC

`taskloop` Construct

Traditional Worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel dgemm in MKL

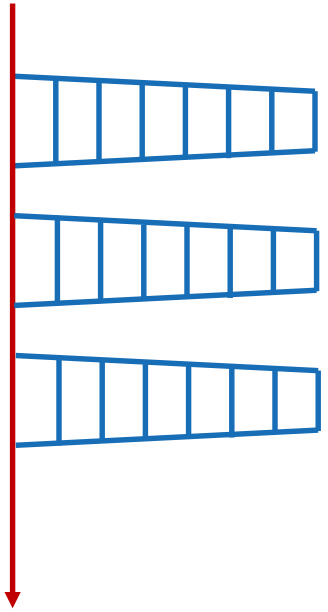
```
void example() {
#pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // dgemm is either parallel or sequential,
        // but has no orphaned worksharing
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);

}   }
```

- Writing such code either
  - → oversubscribes the system,
  - → yields bad performance due to OpenMP overheads, or
  - → needs a lot of glue code to use sequential dgemm only for sub-matrixes

Ragged Fork/Join

- Traditional worksharing can lead to ragged fork/join patterns

```
void example() {
    compute_in_parallel(A);

    compute_in_parallel_too(B);

    cblas_dgemm(…, A, B, …);
}
```



Example: Sparse CG

```
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

```
void matvec(Matrix *A, double *x, double *y) {
    // ...
#pragma omp parallel for \
            private(i,j,is,ie,j0,y0) \
            schedule(static)
for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

The `taskloop` Construct

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

- Syntax (C/C++)

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

- Syntax (Fortran)

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

Clauses for `taskloop` Construct

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

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

- `num_tasks(num-tasks)`
  Create *num-tasks* tasks for iterations of the loop

Example: Sparse CG

```
#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * b
    if (sc->residual <= sc->tole
        break;
    rho_old = rho;
}
```

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

#pragma omp taskloop private(j,is,ie,j0,y0) \
            grain_size(500)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }
    // ...
}
```

Vectorization

Topics

- Exploiting SIMD parallelism with OpenMP
- Using SIMD directives with loops
- Creating SIMD functions

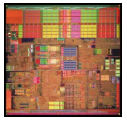
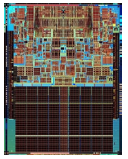
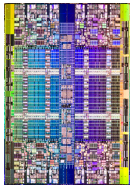
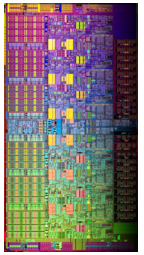
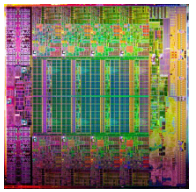
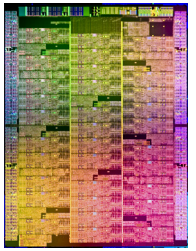
Disclaimer & Optimization Notice

Optimization Notice

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

Notice revision #20110804

Evolution of Intel Hardware

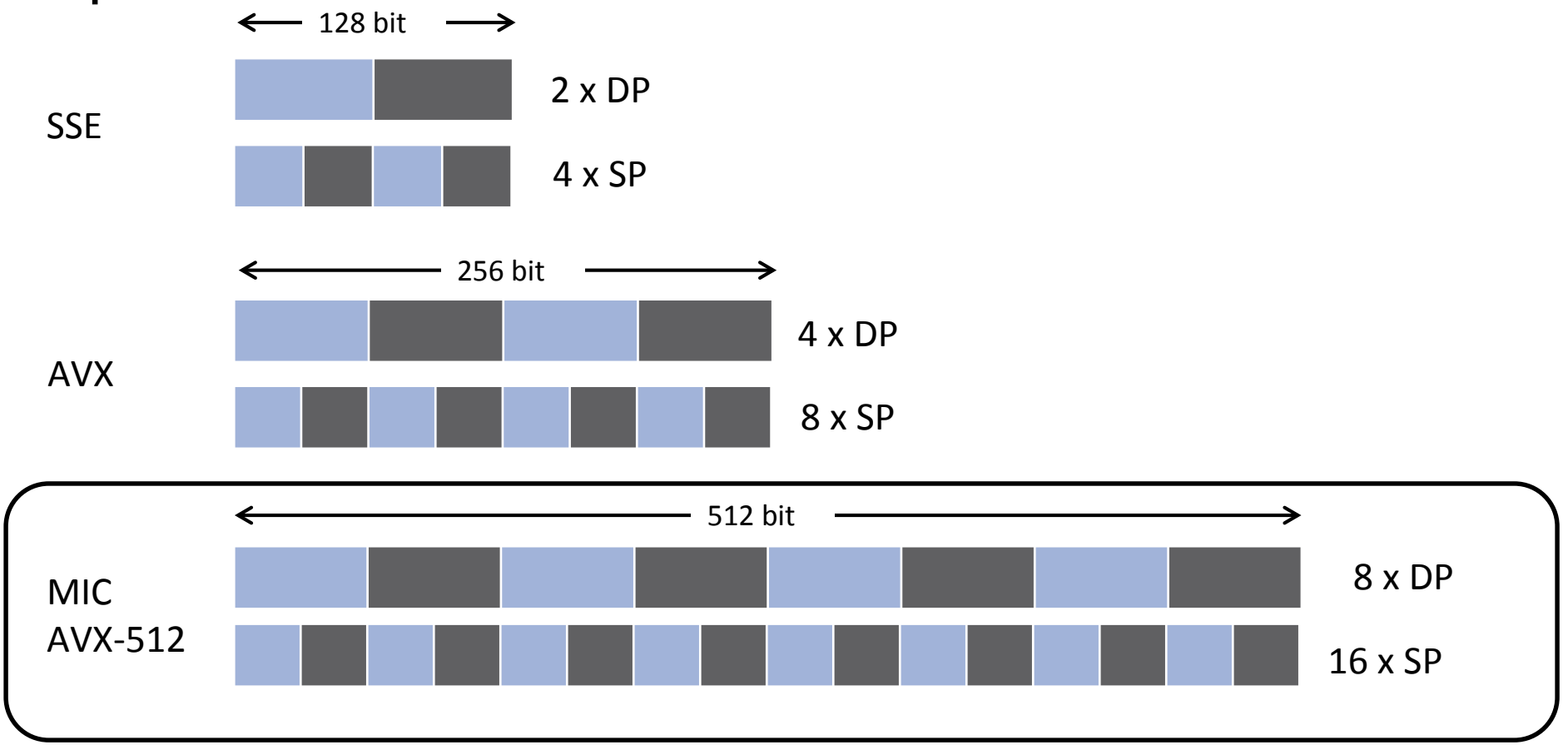







Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v3 series	Intel® Xeon Phi™ Co-processor 7120P
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.3 GHz	1.238 MHz
Core(s)	1	2	4	6	18	61
Thread(s)	2	2	8	12	36	244
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

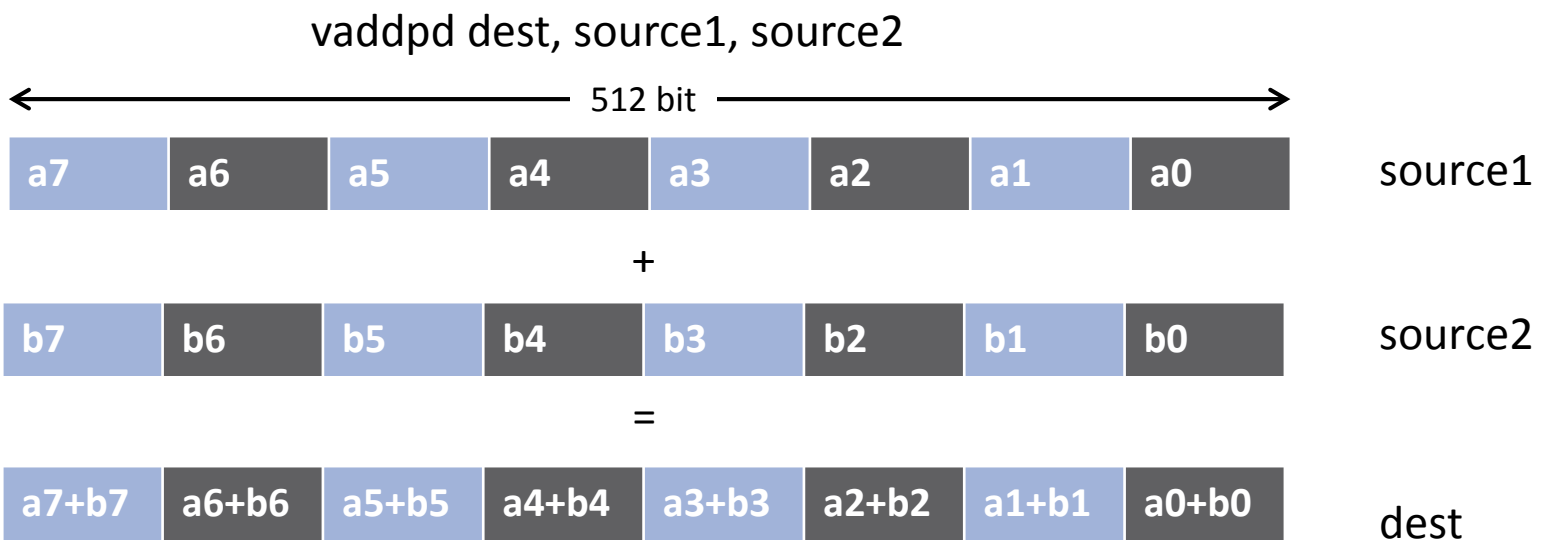
SIMD on Intel® Architecture

- Width of SIMD registers has been growing in the past:



More Powerful SIMD Units

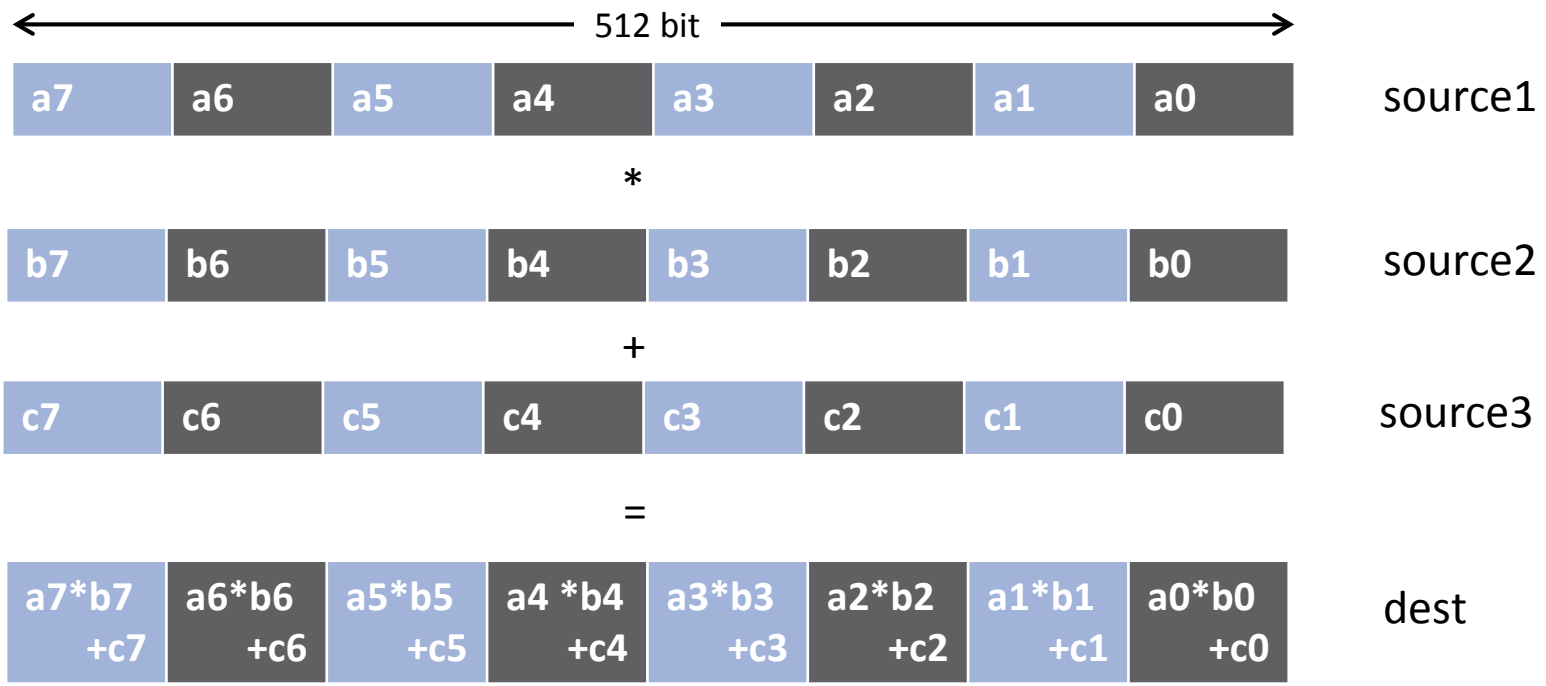
- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



More Powerful SIMD Units

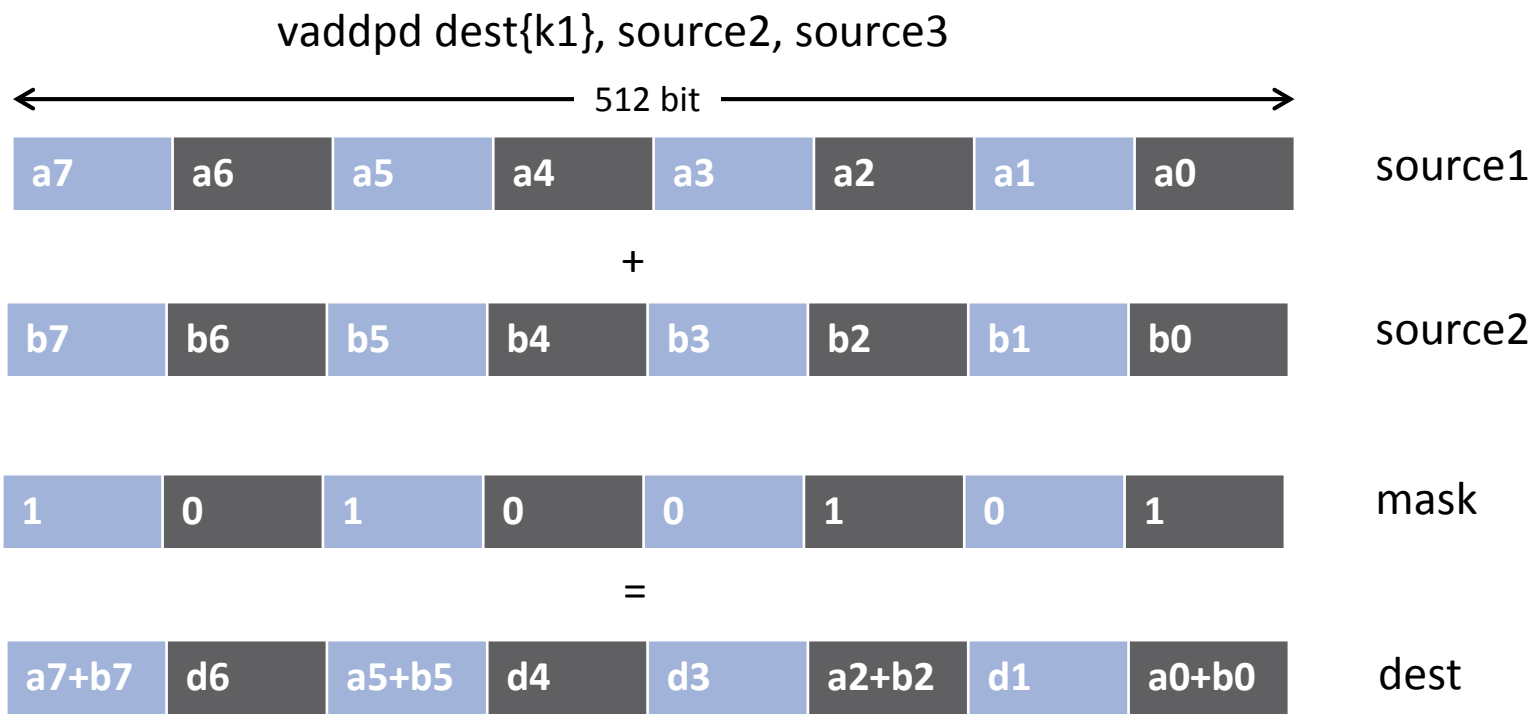
- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor

vfmadd213pd source1, source2, source3



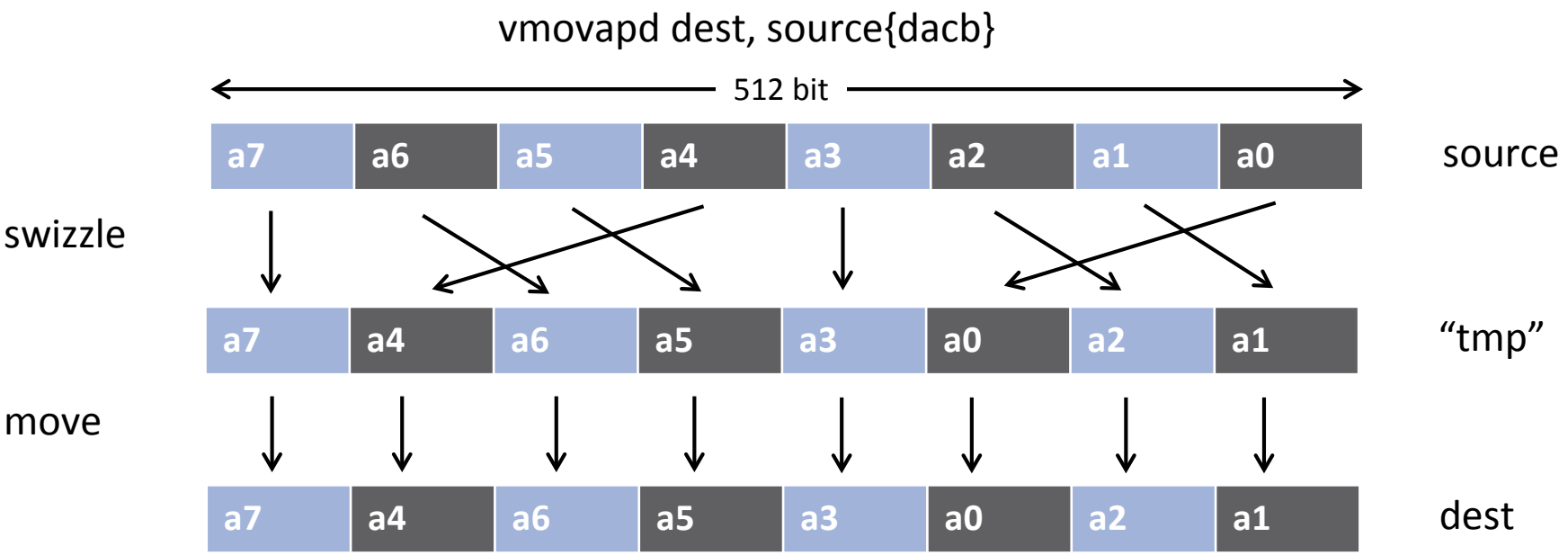
More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is the Intel® Xeon Phi™ Coprocessor



Auto-vectorization

- Auto vectorization only helps in some cases
  - → Increased complexity of instructions make it hard for the compiler to select highest performance instructions
  - → Code pattern needs to be recognized by the compiler
  - → Precision requirements often inhibit SIMD code gen
- Example: Intel® Composer XE

  `-vec` (automatically enabled with **-O3**)

  `-vec-report`

  `-opt-report`

Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - → Alignment
  - → Function calls in loop block
  - → Complex control flow / conditional branches
  - → Loop not "countable"
    - → e.g., upper bound not a runtime constant
  - → Mixed data types
  - → Non-unit stride between elements
  - → Loop body too complex (register pressure)
  - → Vectorization seems inefficient
- Many more … but less likely to occur

Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - → Control-flow dependence
  - → Data dependence
  - → Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW

```
s1: a = 40

    b = 21

s2: c = a + 2
```

ANTI

```
    b = 40

s1: a = b + 1

s2: b = 21
```

Loop-Carried Dependencies

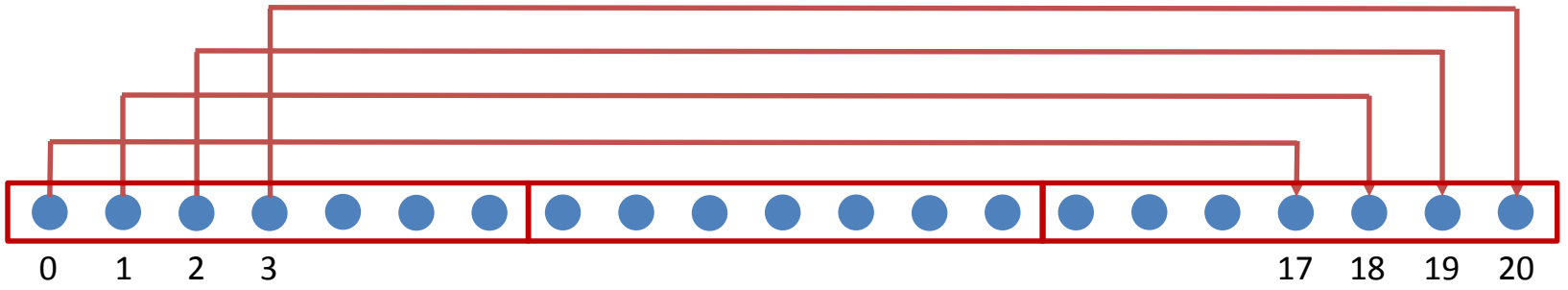
- Dependencies may occur across loop iterations
  - → Loop-carried dependency
- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

- Some iterations of the loop have to complete before the next iteration can run
  - → Simple trick: Can you reverse the loop w/o getting wrong results?

Loop-Carried Dependencies

- Can we parallelize or vectorize the loop?
  - → Parallelization: no
    (except for very specific loop schedules)
  - → Vectorization: yes
    (if vector length is shorter than any distance of any dependency)



Example: Loop not Countable

- "Loop not Countable" plus "Assumed Dependencies"

```
typedef struct {
    float* data;
    size_t size;
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - → Programming models (e.g., Intel® Cilk Plus)
  - → Compiler pragmas (e.g., `#pragma vector`)
  - → Low-level constructs (e.g., `_mm_add_pd()`)

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

You need to trust your compiler to do the "right" thing.

SIMD Loop Construct

- Vectorize a loop nest
  - → Cut loop into chunks that fit a SIMD vector register
  - → No parallelization of the loop body

- Syntax (C/C++)

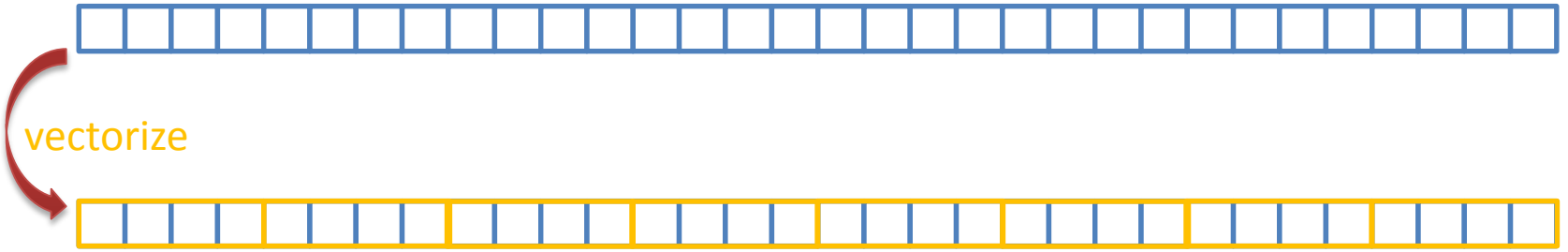
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

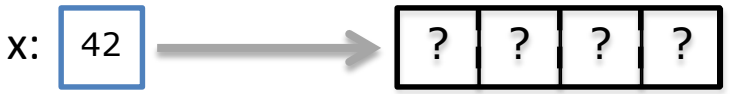
Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

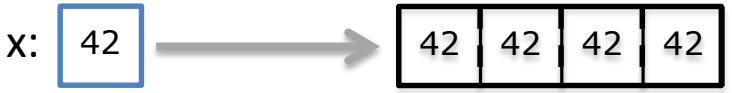


Data Sharing Clauses

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



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



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

SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → In practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture
- `collapse (n)`

SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - → Distribute a loop's iteration space across a thread team
  - → Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

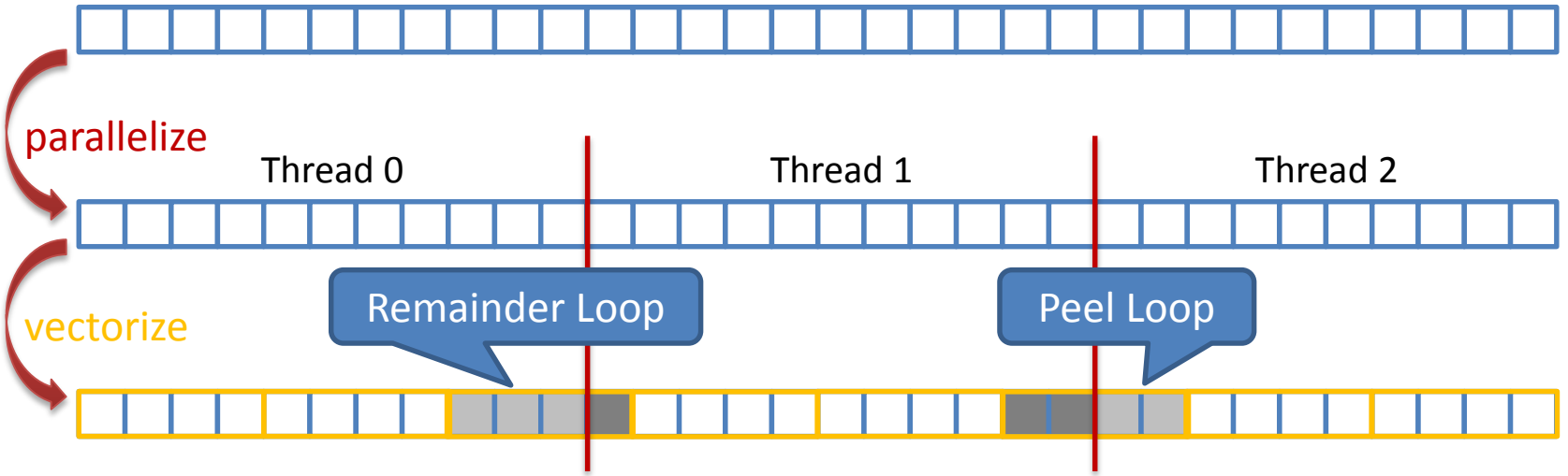
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Be Careful What You Wish For…

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance
- In the above example …
  - → and AVX2, the code will only execute the remainder loop!
  - → and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(simd: static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- Chooses chunk sizes that are multiples of the SIMD length
  - → First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
  - → Remainder loops are not triggered
  - → Likely better performance

SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

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

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

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

SIMD Function Vectorization

- `simdlen (length)`
  - → generate function to support a given vector length
- `uniform (argument-list)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement
- `linear (argument-list[:linear-step])`
- `aligned (argument-list[:alignment])`

Same as before

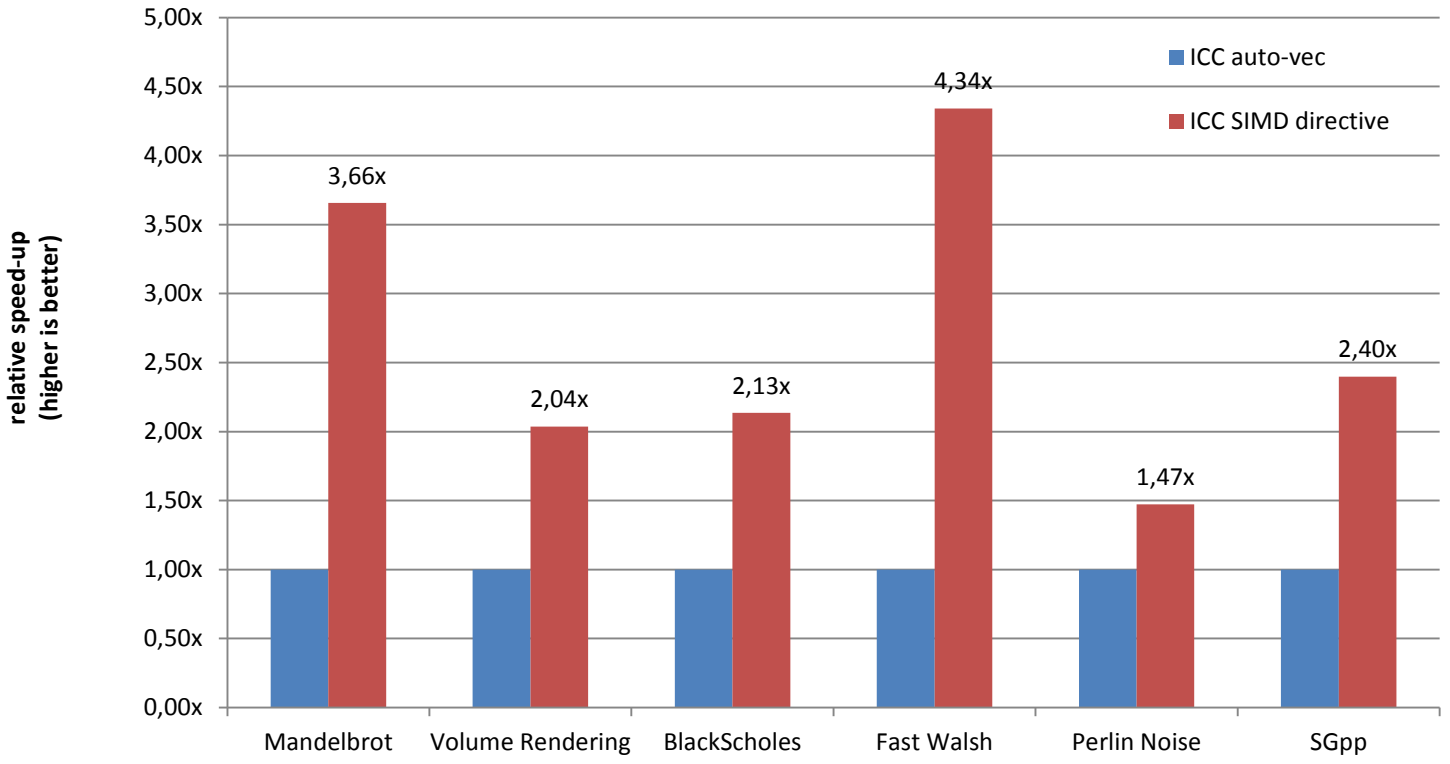
inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

SIMD Constructs & Performance



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

NUMA

OpenMP and Performance

- Two of the more obscure things that can negatively impact performance are cc-NUMA effects and false sharing

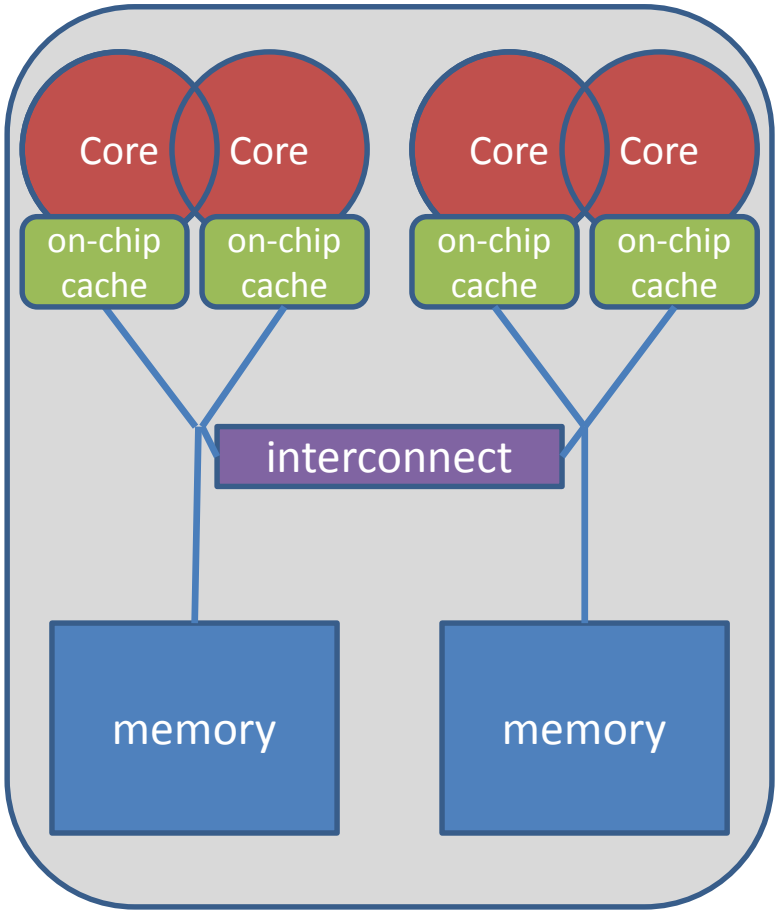
- ***Neither of these are restricted to OpenMP***
  - → But they most show up because you used OpenMP
  - → In any case they are important enough to cover here

Non-uniform Memory

How To Distribute The Data ?

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

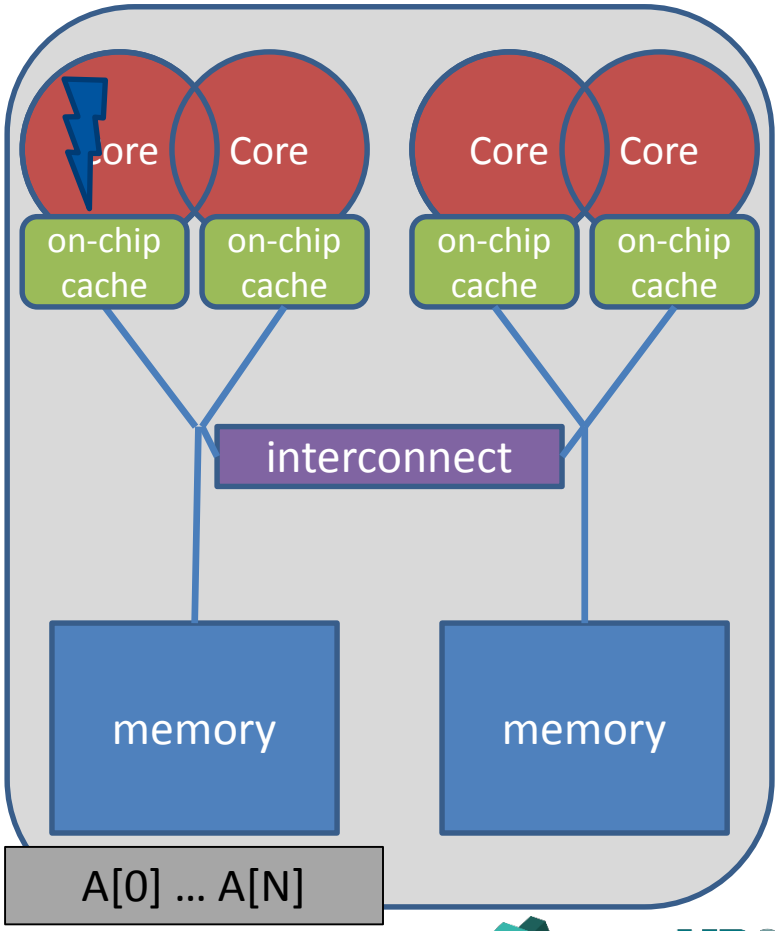
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)**

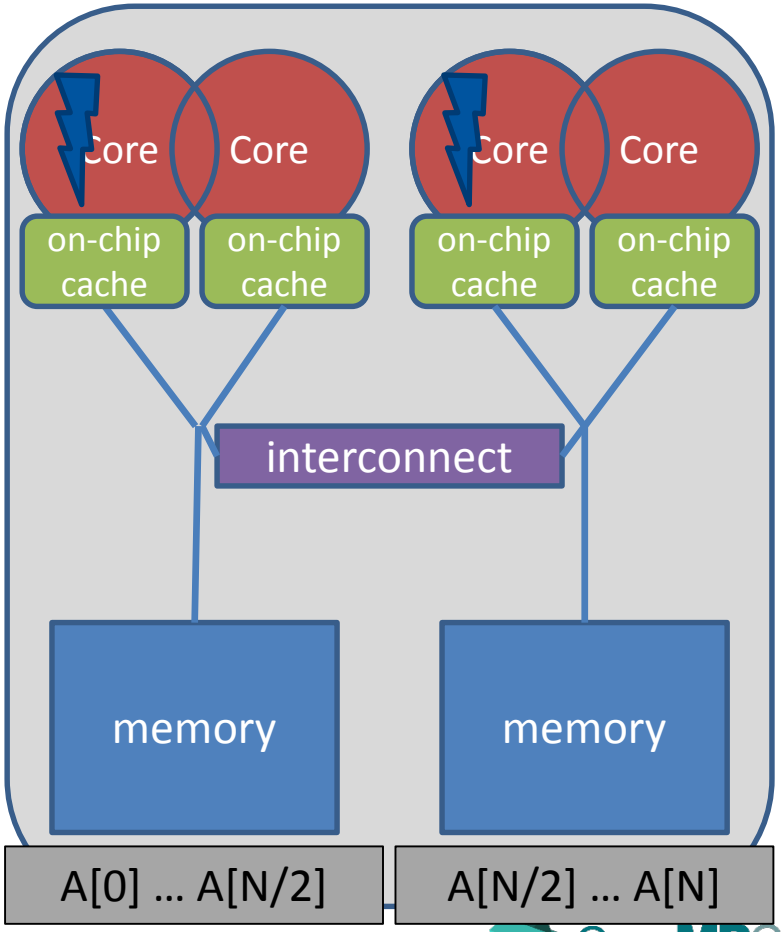
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- **First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```

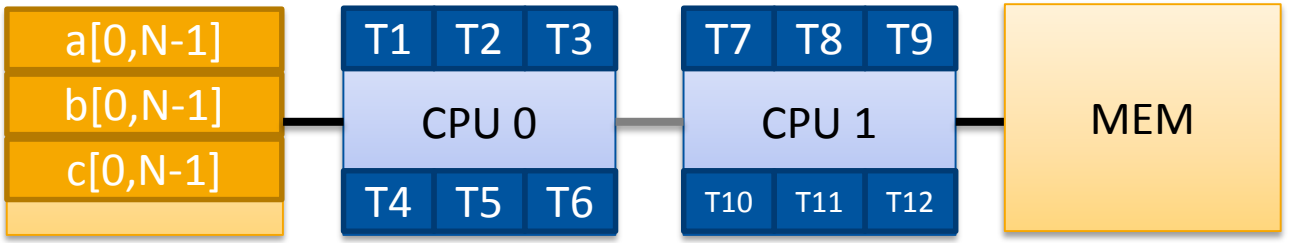


Serial vs. Parallel Initialization

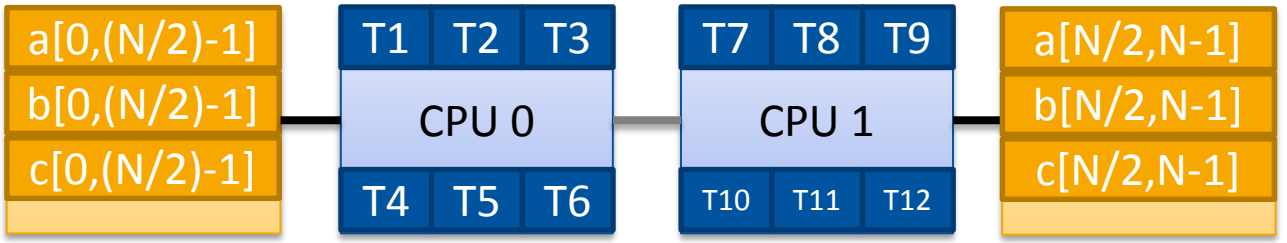
- **Stream example with and without parallel initialization.**
  - → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

ser_init:



par_init:



Thread Binding and Memory Placement

Get Info on the System Topology

- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**
  - → Intel MPI's `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor IDs to CPU cores used by the OS
  - → hwlocs' `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor IDs to CPU cores used by the OS and additional information on caches

Decide for Binding Strategy

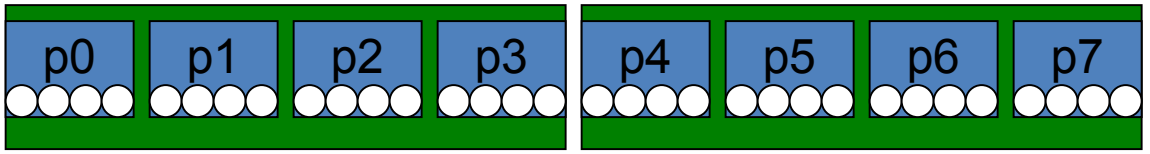
- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - → Putting threads far apart, i.e., on different sockets
    - → May improve the aggregated memory bandwidth available to your application
    - → May improve the combined cache size available to your application
    - → May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - → May improve performance of synchronization constructs
    - → May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

OpenMP 4.0: Places + Policies

- **Define OpenMP places**
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e., `OMP_PLACES=cores`
- **Define a set of OpenMP thread affinity policies**
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- **Goals**
  - → user has a way to specify where to execute OpenMP threads for locality between OpenMP threads / less false sharing / memory bandwidth

OMP_PLACES env. variable

- Assume the following machine:



→2 sockets, 4 cores per socket, 4 hyper-threads per core

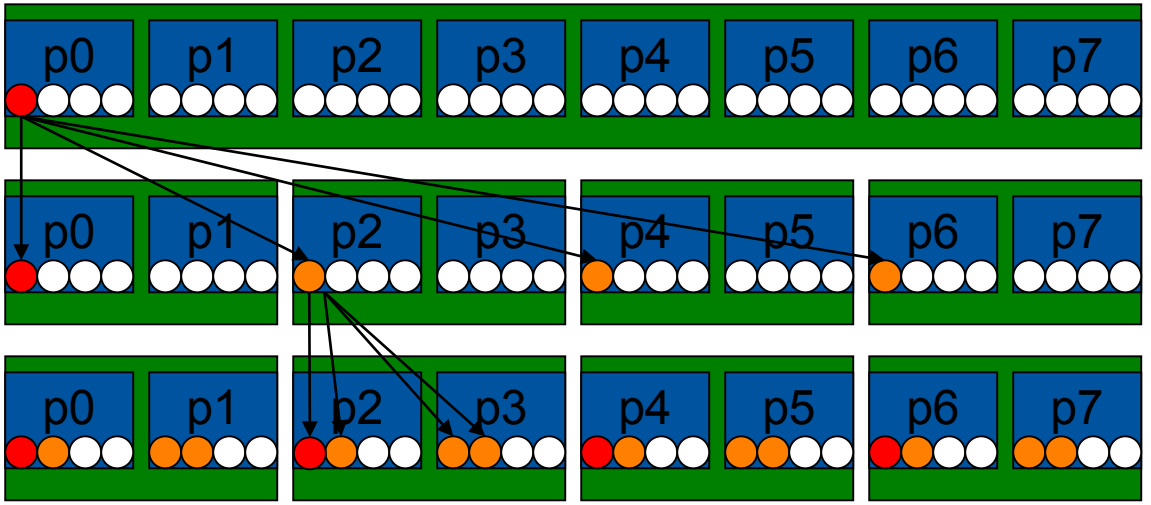
- Abstract names for OMP_PLACES:
  - →threads: Each place corresponds to a single hardware thread on the target machine.
  - →cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - →sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

OpenMP 4.0: Places + Binding Policies (2/2)

- **Example's Objective:**
  - → separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)**
  - → spread creates partition, compact binds threads within respective partition

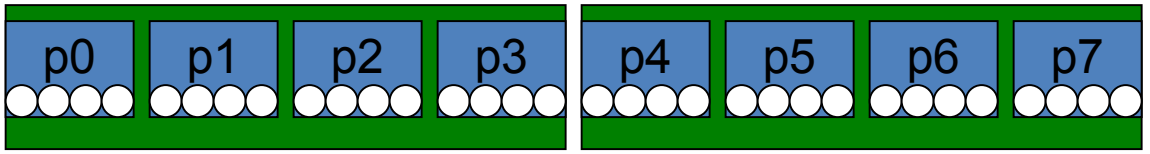
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4    = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- **Example**
  - → initial
  - → spread 4
  - → close 4



More Examples (1/3)

- Assume the following machine:

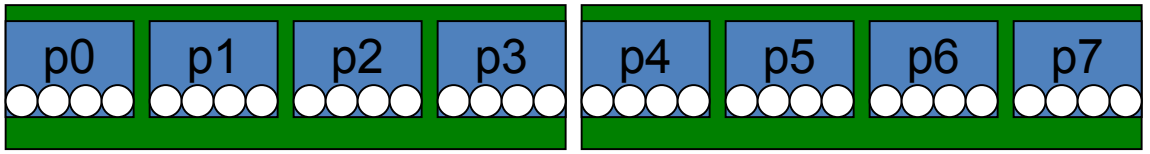


  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Parallel Region with two threads, one per socket
  → `OMP_PLACES=sockets`
  → `#pragma omp parallel num_threads(2) \ proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



- Parallel Region with four threads, one per core, but only on the first socket
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(4) \ proc_bind(close)`

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(2) \ proc_bind(spread)`
  - → `#pragma omp parallel num_threads(4) \ proc_bind(close)`

Places API (1/2)

- 1: Query information about binding and a single place of all places with ids 0 … `omp_get_num_places()`:
- `omp_proc_bind_t omp_get_proc_bind()`: returns the thread affinity policy (omp_proc_bind_false, true, master, …)
- `int omp_get_num_places()`: returns the number of places
- `int omp_get_place_num_procs(int place_num)`: returns the number of processors in the given place
- `void omp_get_place_proc_ids(int place_num, int* ids)`: returns the ids of the processors in the given place

Places API (2/2)

- 2: Query information about the place partition:
- `int omp_get_place_num()`: returns the place number of the place to which the current thread is bound
- `int omp_get_partition_num_places()`: returns the number of places in the current partition
- `void omp_get_partition_place_nums(int* pns)`: returns the list of place numbers corresponding to the places in the current partition

Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

A First Summary

- Everything under control?
- In principle Yes, but only if
  - → threads can be bound explicitly,
  - → data can be placed well by first-touch, or can be migrated,
  - → you focus on a specific platform (= os + arch) → no portability
- What if the data access pattern changes over time?
- What if you use more than one level of parallelism?

NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e., Linux >= 2.4) determine the physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU that causes the page fault
- Explicit Migration: Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall
- Next Touch: The binding of pages to NUMA nodes is removed and pages are put in the location of the next „touch“; well supported in Solaris, expensive to implement in Linux
- Automatic Migration: No support for this in current operating systems

User Control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - → By carefully touching data by the thread which later uses it
  - → By changing the default memory allocation strategy
    - → Linux: `numactl` command
  - → By explicit migration of memory pages
    - → Linux: `move_pages()`

- Example: using numactl to distribute pages round-robin:
  - → `numactl -interleave=all ./a.out`

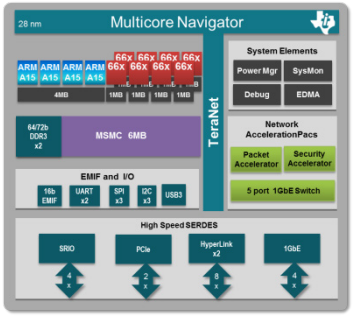
Offload Programming

Topics

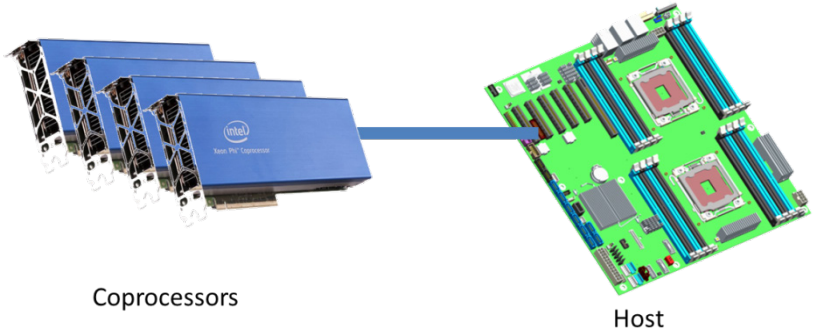
- Heterogeneous device execution model
- Mapping variables to a device
- Accelerated workshare

Device Model

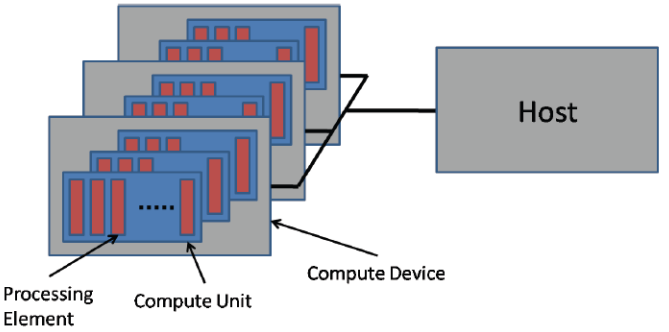
- OpenMP 4.0 supports heterogeneous systems
- Device model:
  - → One host device and
  - → One or more target devices



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 offloads **target** regions to target devices.

OpenMP 4.0 Device Constructs

- Execute code on a target device
  - **omp target** *[clause[[,] clause],…]*
    structured-block
  - **omp declare target**
    [function-definitions-or-declarations]
- Map variables to a target device
  - **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);`
- Environment variable
  - Control default device through **`OMP_DEFAULT_DEVICE`**
  - Accepts a non-negative integer value

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 other OpenMP tasks 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

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

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(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);
}
```

```
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
```

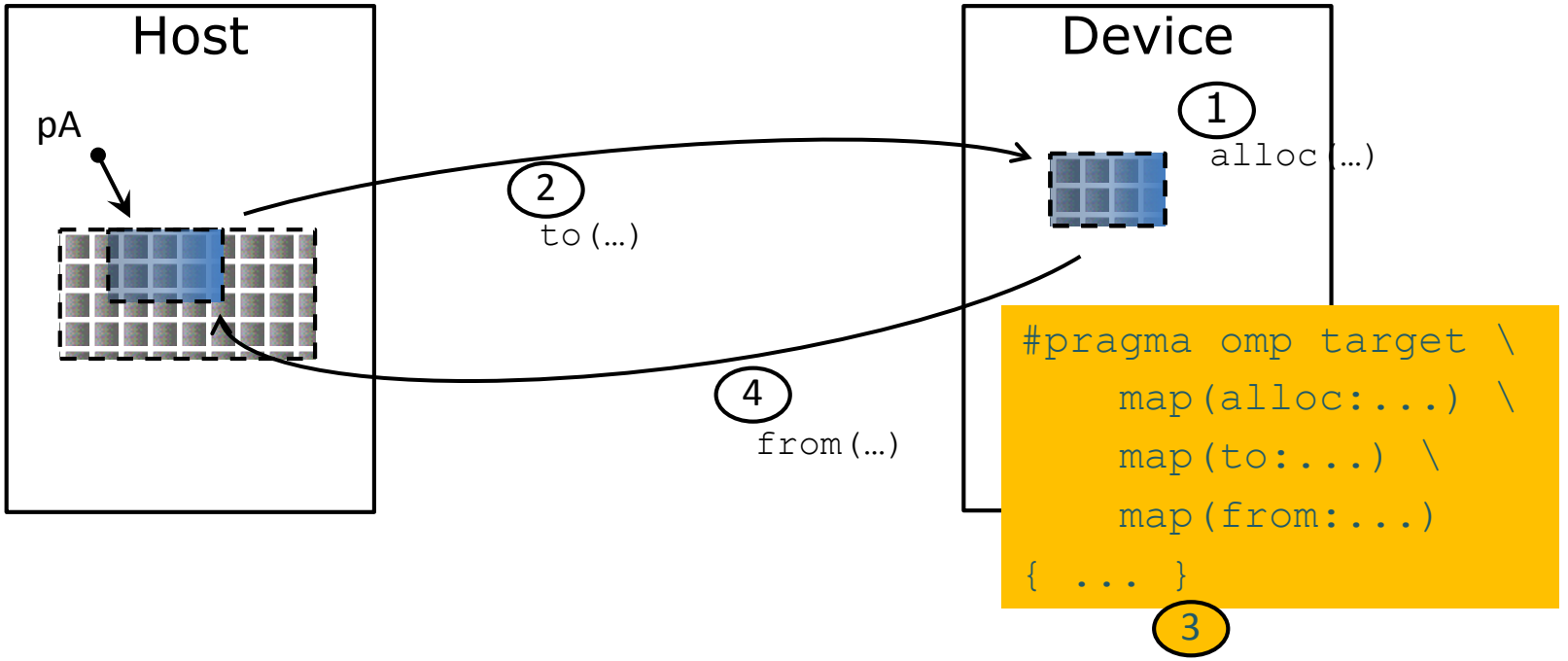
- The array sections for v1, v2, and p are explicitly *mapped* into the device data environment.
- The variable N is implicitly *mapped* into the device data environment

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.



`map` Clause

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

  `map(`*`[map-type:] list`*`)`
- Where map-type is:
  - → `alloc:` allocate storage for corresponding variable
  - → `to:` alloc and assign value of original variable to corresponding variable on entry
  - → `from:` alloc and assign value of corresponding variable to original variable on exit
  - → `tofrom:` default, both to and form

map Clause

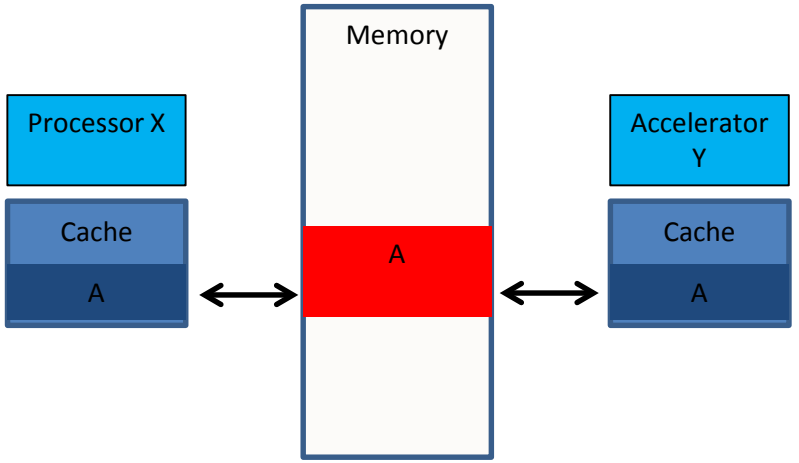
```
extern void init(float*, float*, int);
extern void output(float*, int);

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
```

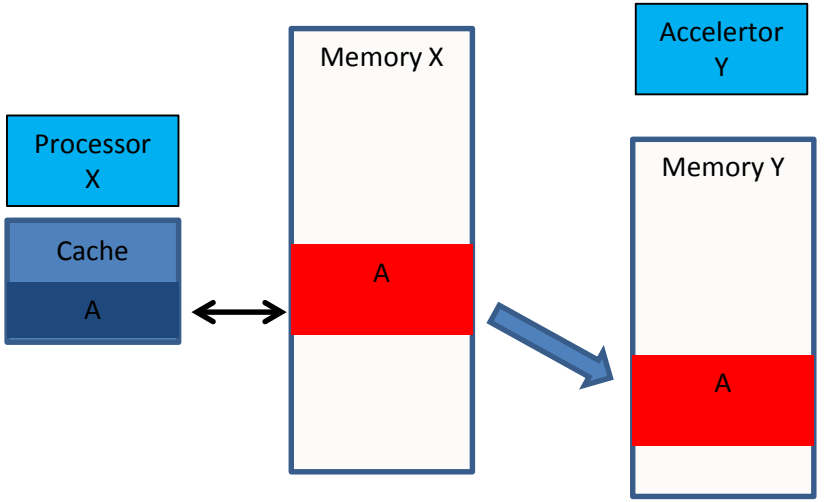
- On entry to the target region:
  - Allocate corresponding variables v1, v2, and p in the device data environment.
  - Assign the corresponding variables v1 and v2 the value of their respective original variables.
  - The corresponding variable p is undefined.
- On exit from the target region:
  - Assign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



Distributed 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.

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.

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)
  }
```

host
target
host
target
host

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)
```

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 device(0) 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)
  }
```

host
target
host
target
host

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)
```

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

target declare 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)
```

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
```

Review: Loop worksharing

Sequential code

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

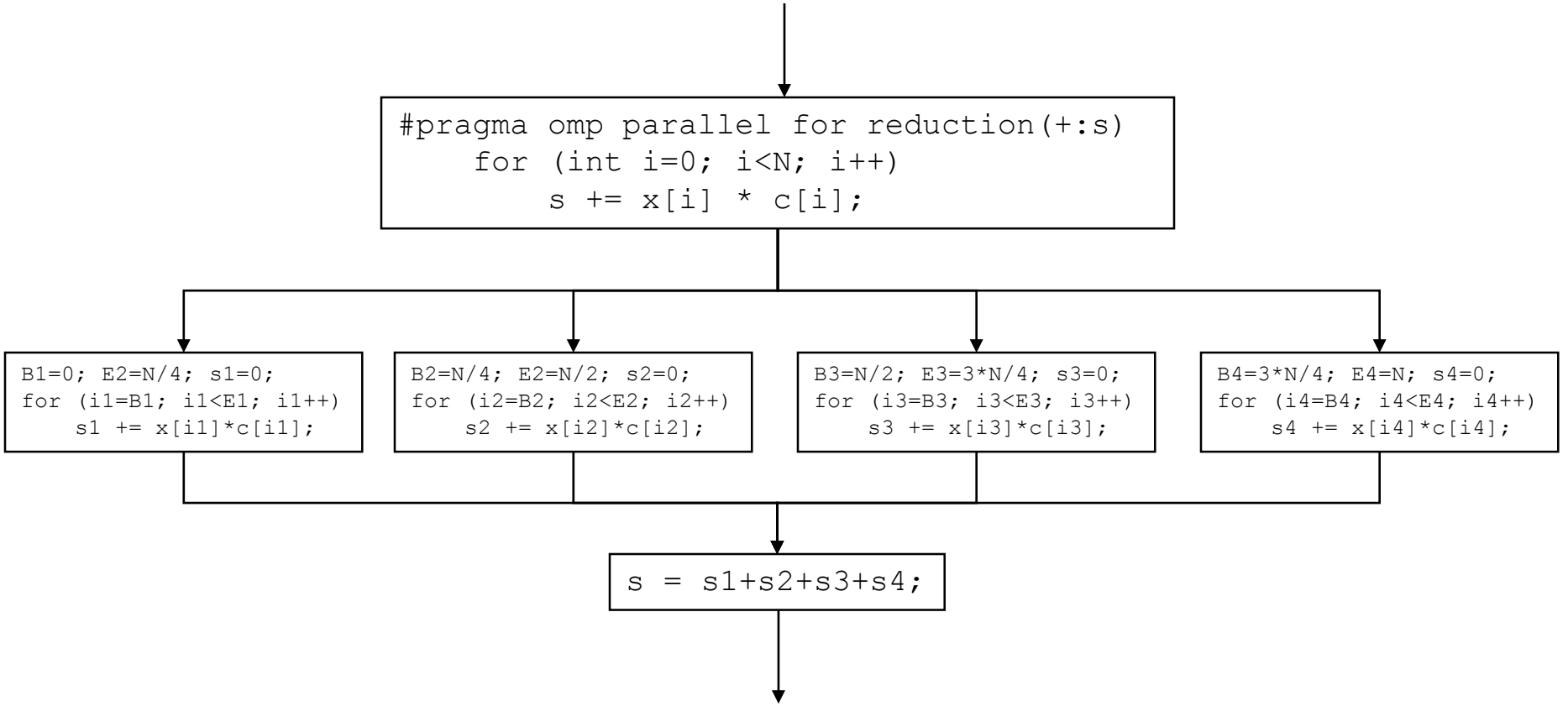
OpenMP Parallel Region

```
#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 the 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

- New kind of worksharing construct
  - → 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)
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:n,a,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.

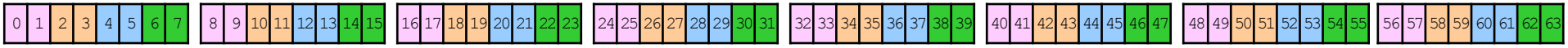
Distribute Parallel SIMD

64 iterations assigned to 2 teams; Each team has 4 threads; Each thread has 2 simd lanes

Distribute Iterations across 2 teams

In a team workshare iterations across 4 threads

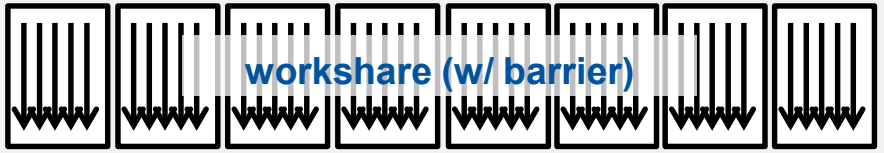
In a thread use simd parallelism



SAXPY: Accelerated worksharing

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

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

#pragma omp target map(to:n,a,x[:n]) map(y[:n])
#pragma omp teams distribute
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]);
  }
```

New in OpenMP 4.5

- Unstructured Data Mapping
  - → `target enter data map(list)`
  - → `target exit data map(list)`
- Asynchronous Execution
  - → target regions are now task regions
  - → `target depend(in: x) nowait`
- Performance oriented changes
  - → Scalar variables are firstprivate
  - → Improvements for Array sections in C/C++
- New device Memory API routines
  - → `omp_target_free()`, `omp_target_alloc()`, etc…
- Support for device pointers
  - → `target data use_deviceptr(a)`
  - → `target is_deviceptr(p)`
- Mapping structure elements
  - → `target map(S.x, s.y[:N])`
- New combined constructs
  - → `target parallel` …
- New ways to map global variables
  - → `declare target link(x)`
- Extensions to the if clause for combined/composite constructs
  - → `target parallel if(target,c1) if(parallel, c2)`

OpenMP 4.5 – Asynchronous Execution

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:

```
if([ target :] scalar-expression)
device(integer-expression)
private(list)
firstprivate(list)
map([[map-type-modifier[,]] map-type: ] list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

- The **nowait** clause indicates that the encountering thread does not wait for the target region to complete.
- A host task is generated that encloses the target region.
- The **depend** clause can be used for synchronization with other tasks

OpenMP 4.5 – unstructured data mapping

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

```
#pragma omp target enter data [clause[[,] clause]...] new-line
```

where *clause* is one of the following:

```
if([ target enter data :] scalar-expression)
device(integer-expression)
map([ [map-type-modifier[,]] map-type : ] list)
depend(dependence-type : list)
nowait
```

```
#pragma omp target exit data [clause[[,] clause]...] new-line
```

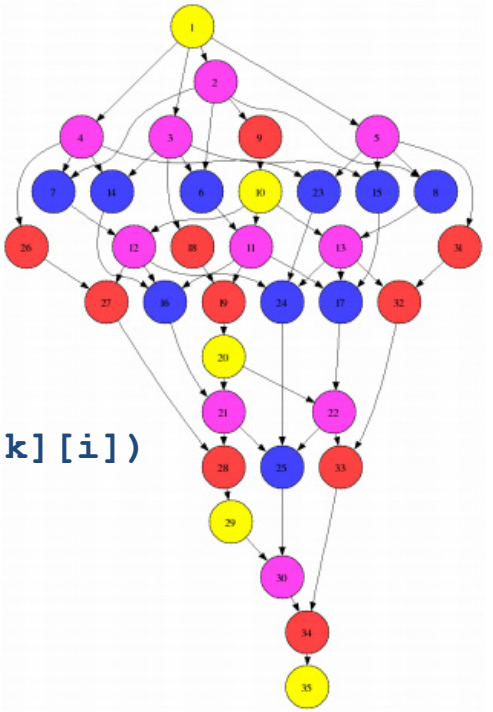
where *clause* is one of the following:

```
if([ target exit data :] scalar-expression)
device(integer-expression)
map([ [map-type-modifier[,]] map-type : ] list)
depend(dependence-type : list)
nowait
```

- Structured **target data** construct is too restrictive and does not fit for C++ (de)constructors.
- **target enter data**
  - Map variable to a device
- **target exit data**
  - Map variable from a device

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]);
       }
   }
}
```



* image from BSC

Tasks and Target 4.5 Example

```
#pragma omp declare target
#include <stdlib.h>
#include <omp.h>
extern void compute(float *, float *, int);
#pragma omp end declare target

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

   #pragma omp target enter data map(alloc: v1[:N], v2[:N])

   #pragma omp target nowait depend(out: v1, v2)
   {
      compute(v1,v2,N);
   }

   #pragma omp task
   other_work(); // execute asynchronously on host device

   #pragma omp target map(from:p[0:N]) nowait depend(in: v1, v2)
   {
      #pragma omp distribute parallel for
      for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];
   }

   #pragma taskwait

   #pragma omp target exit data map(release: v1[:N], v2[:N])
}
```

Performance Oriented Changes

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:

```
if([ target :] scalar-expression)
device(integer-expression)
private(list)
firstprivate(list)
map([[map-type-modifier[,]] map-type: ] list)
is_device_ptr(list)
defaultmap(tofrom:scalar)
nowait
depend(dependence-type: list)
```

- By default scalar variables are now firstprivate
- Use **defaultmap** clause to change default behavior for scalars
- By default pointer variables in array sections are private

Device pointer

- **New clauses**
  - **#pragma omp target data … use_device_ptr(list) ..**
  - **#pragma omp target … is_device_ptr(list) ..**

- **New API**
  - **void* omp_target_alloc(size_t** *size*, **int** *device_num*);
  - **void omp_target_free(void *** *device_ptr*, **int** *device_num*);
  - **int omp_target_is_present(void *** *ptr*, **size_t** *offset*, **int** *device_num*);
  - **int omp_target_memcpy(void *** *dst*, **void *** *src*, **size_t** *length*, **size_t** *dst_offset*, **size_t** *src_offset*, **int** *dst_device_num*, **int** *src_device_num*);
  - **int omp_target_memcpy_rect(void *** *dst*, **void *** *src*, **size_t** *element_size*, **int** *num_dims*, **const size_t*** *volume*, **const size_t*** *dst_offsets*, **const size_t*** *src_offsets*, **const size_t*** *dst_dimensions*, **const size_t*** *src_dimensions*,**int** *dst_device_num*, **int** *src_device_num*);
  - **int omp_target_associate_ptr(void *** *host_ptr*, **void *** *device_ptr*, **size_t** *size*, **size_t** *device_offset*, **int** *device_num*);
  - **int omp_target_disassociate_ptr(void *** *ptr*, **int** *device_num*);
  - **int omp_get_initial_device (void)**

Combined Constructs

- #pragma omp target parallel
- #pragma omp target parallel for
- #pragma omp target parallel for simd
- #pragma omp target simd

Declare target link

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

where *clause* is one of the following:

*[***to***]* **(***extended-list***)**

link(*list***)**

The list items of a **link** clause are not mapped by the **declare target** directive. Instead, their mapping is deferred until they are mapped by **target data** or **target** constructs. They are mapped only for such regions.

```
#pragma omp declare target
int foo();
#pragma omp end declare target

extern int a;
#pragma omp declare target link(a)

int foo()
{
   a = 1;
}
main()
{
   #pragma omp target map(a)
   {
       foo();
   }
}
```

Clause changes (1)

- **If clause**
  - **if(***[directive-name-modifier* **:]** *scalar-expression***)**

The effect of the **if** clause depends on the construct to which is it applied. For combined or composite constructs, the **if** clause only applies to the semantics of the construct named in the directive-name-modifier if one is specified. If no directive-name-modifier is specified for a combined or composite construct then the **if** clause applies to all constructs to which an if clause can apply.

Eg:-

```
#pragma omp target data if(target data : 1)
#pragma omp task if(task : 1)
#pragma omp target parallel for if( target : 1 ) if ( parallel: 0 )
```

`if` Clause Example

```
#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.

The End…

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