A short stroll along the road of OpenMP* 4.0

Christian Terboven <terboven@itc.rwth-aachen.de>
Michael Klemm <michael.klemm@intel.com>

Members of the OpenMP Language Committee

Our roles for today

Michael "the master of desaster" Klemm, works for a company that builds very fast multi-core processors and knows how to optimize codes for these. However sometimes these processors do not run programs as fast or scalable as hoped. Today, Michael has to tell us why.

Christian "the excited user" Terboven works a lot with engineers and scientists in parallelizing their codes, and was involved in the design of several OpenMP extensions. Today, Chris will explain how to use OpenMP *correctly* where some say it is not applicable.

Legal Disclaimer & Optimization Notice

Optimization Notice

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

Notice revision #20110804

OpenMP Misbeliefs

- OpenMP does not scale well
- OpenMP is only good for simple loops (Worksharing)
- OpenMP-parallel code is not elegant
- Shared memory parallelization is only about cores
- OpenMP does not work well for C++ codes
- For accelerators you have to use CUDA* or OpenCL*

Agenda

- Structure of Data in Memory
- Vectorization / SIMD Parallelism
- Tasking
- Accelerators

Structure of Data in Memory

Example: Bounding Box Code

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

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

- **„Problems“ for an OpenMP parallelization?**

  → Reduction operation has to work with non-POD datatypes

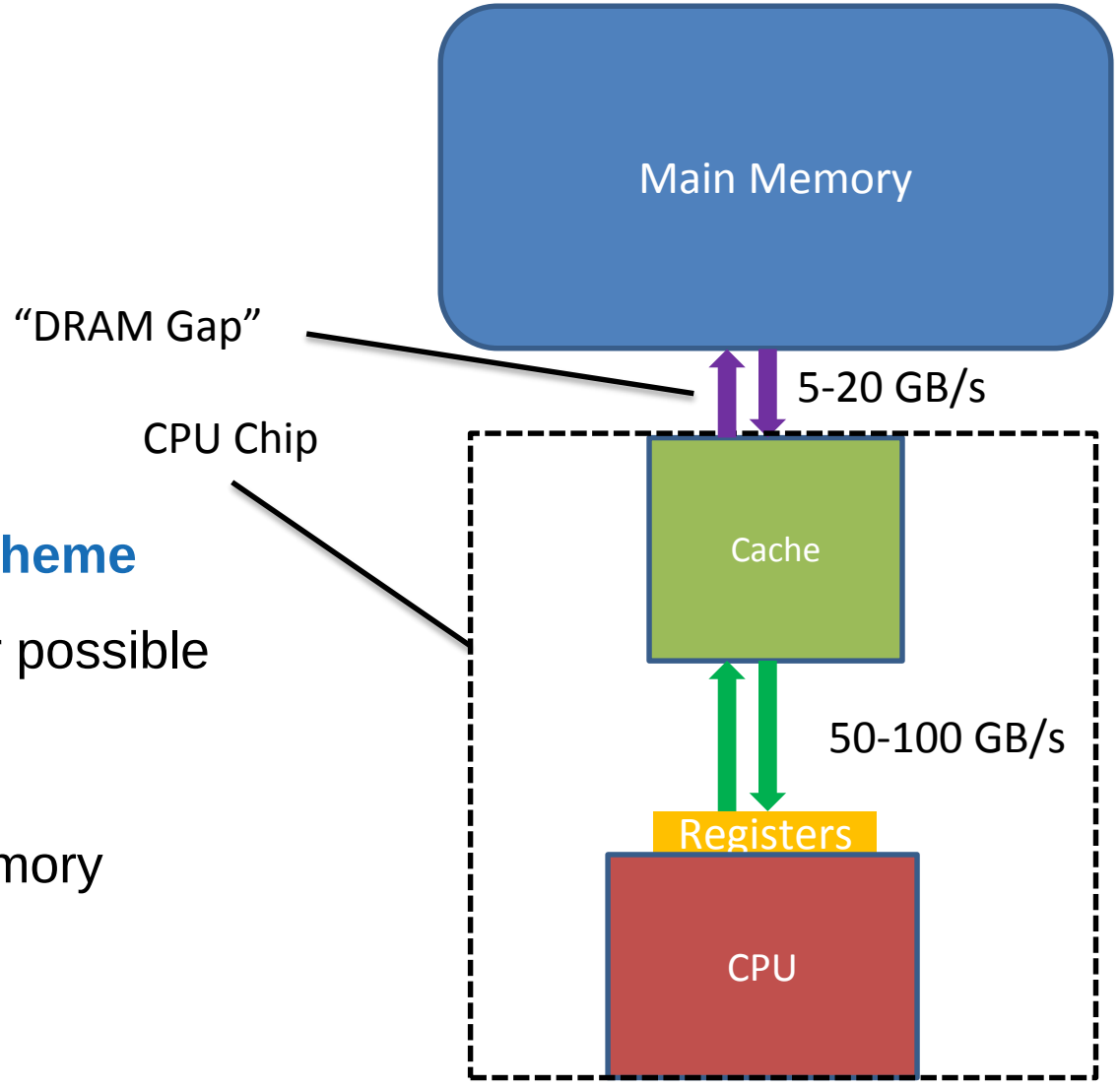
  → Loop employs C++ iterator over std::vector datatype elements

OpenMP

Except from ugly code – why does this code run so slow?

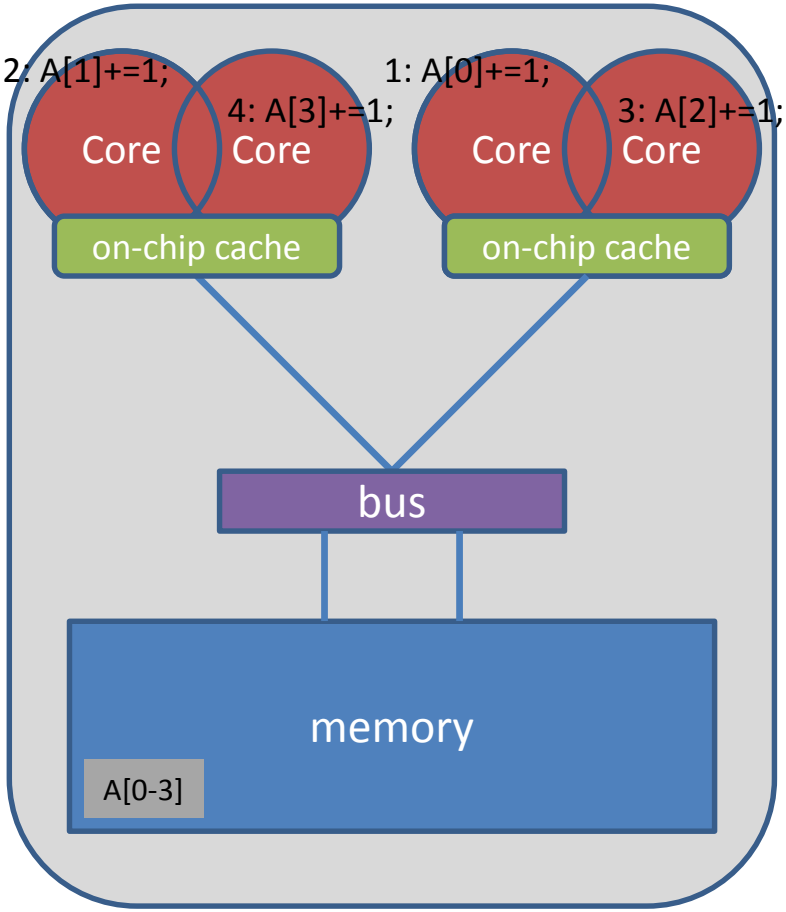
Memory Hierarchy

- In modern computer design memory is divided into different levels:
- Registers
- Caches
- Main Memory
- Access follows the scheme
  - → Registers whenever possible
  - → Then the cache
  - → At last the main memory



False Sharing

- **False Sharing: Parallel accesses to the same cache line may have a significant performance impact!**



Caches are organized in lines of typically 64 bytes: integer array a[0-4] fits into one cache line.

Whenever one element of a cache line is updated, the whole cache line is Invalidated.

Local copies of a cache line have to be re-loaded from the main memory and the computation may have to be repeated.

Bounding Box w/ OpenMP 4.0

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

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

- **OpenMP 4.0 brings user-defined reductions**

→ *name*: `minp`, *datatype*: `Point2D`

→ *read*: `omp_in`, *written to*: `omp_out`, *initialization*: `omp_priv`

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

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

OpenMP

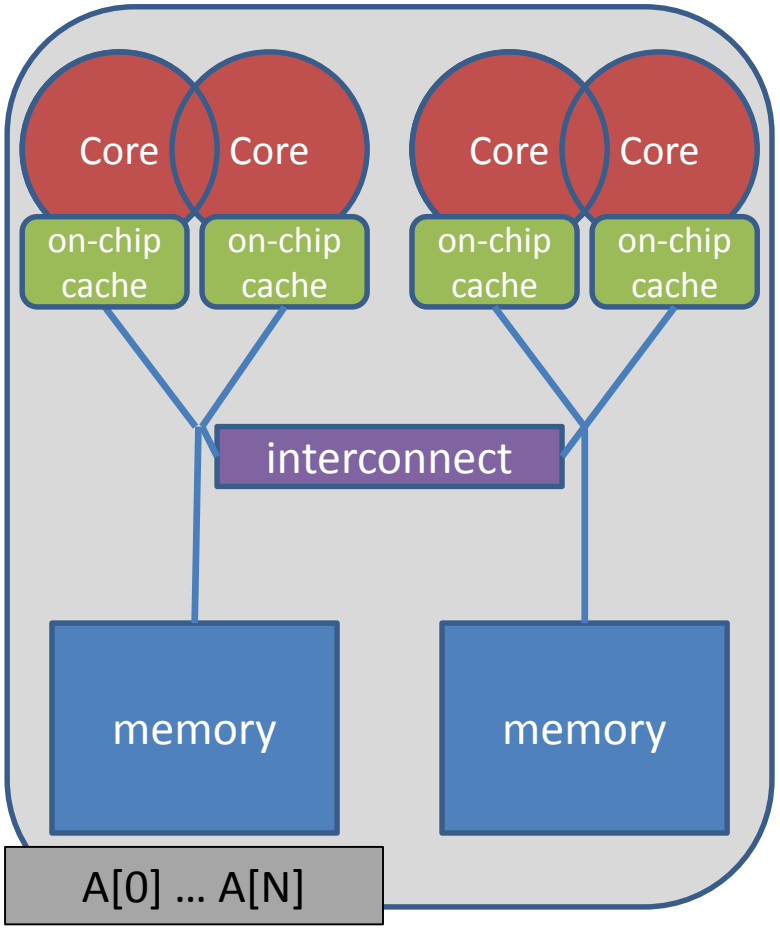
BUT: what if the point cloud is really big?

Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread**

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

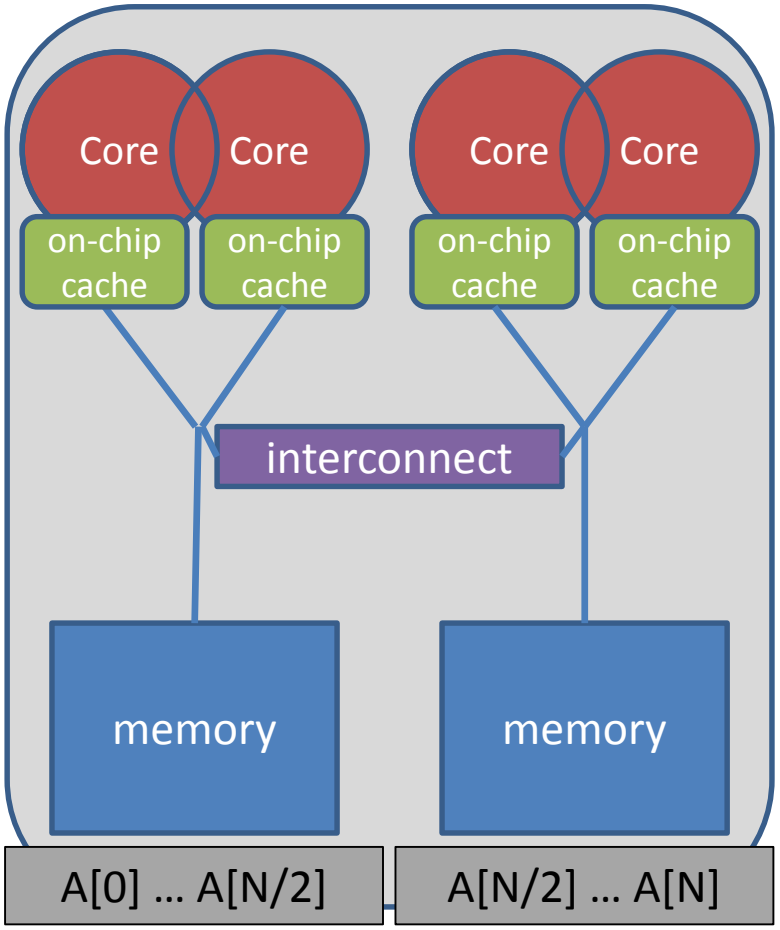
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

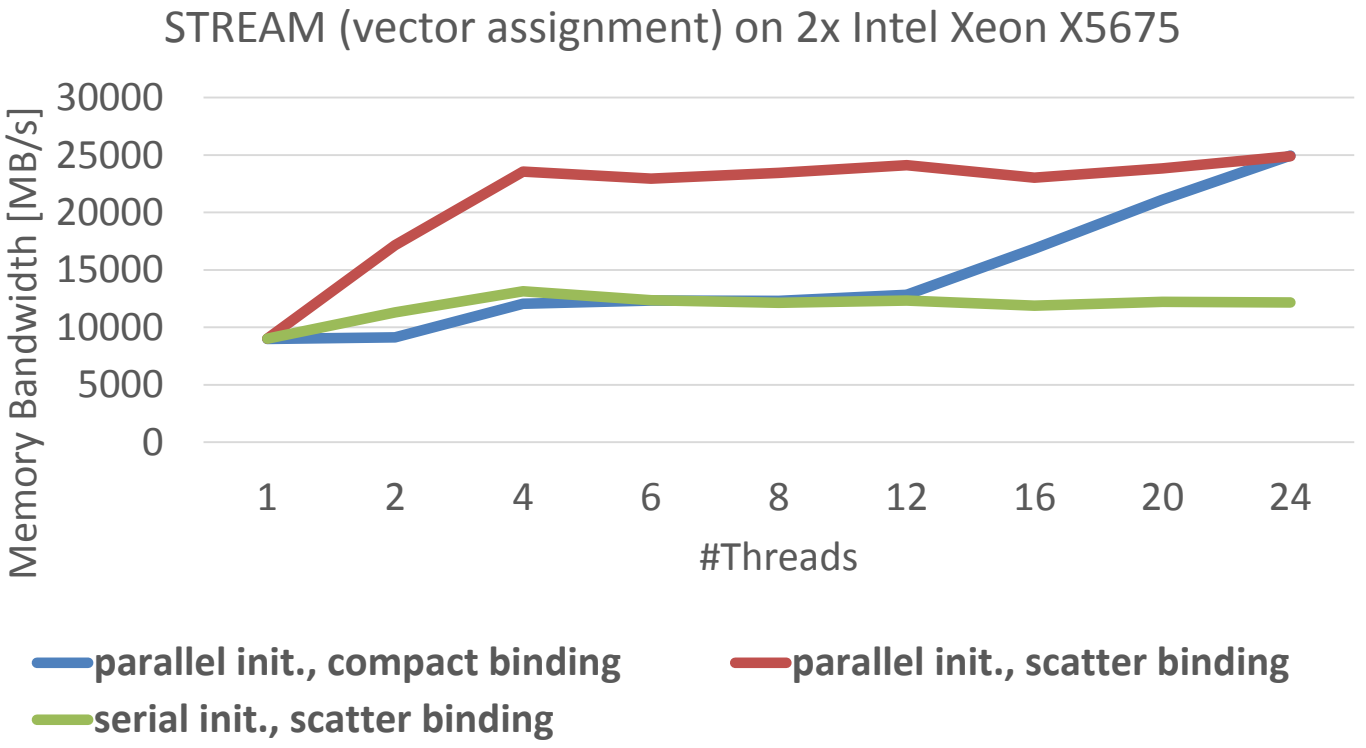
- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```
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

- **Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:**



Bounding Box w/ OpenMP: ☺

- **Employ first-touch + thread binding in the shell:**

```
export OMP_PLACES=cores
```

as an initialization (vector always performs default initialization):

```
std::valarray<Point2D> points(NUM_POINTS);

#pragma omp parallel for proc_bind(spread)
for (int i = 0; i < NUM_POINTS; i++) {
        float x = RANGE / rand();
        float y = RANGE / rand();
        points[i] = Point2D(x, y);
}
```

or with std::vector and appropriate allocator:

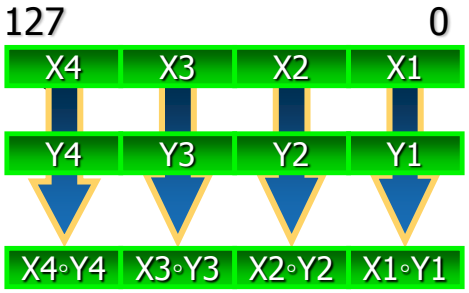
```
std::vector<Point2D, no_init_allocator> points(NUM_POINTS);
```

OpenMP

Ok, thanks. But, how can I get even more performance from a chip?

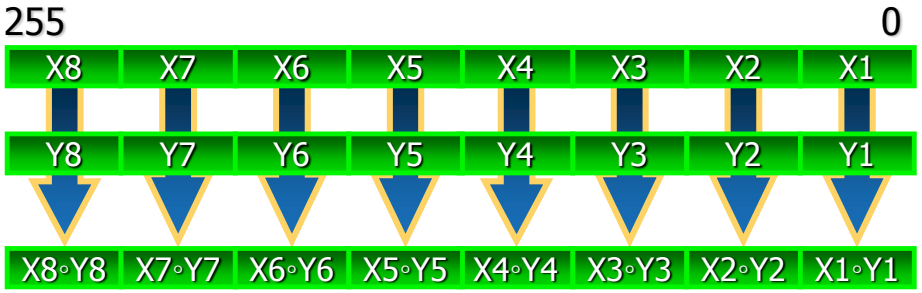
Vectorization / SIMD Parallelism

SIMD is Here to Stay



SSE
Vector size: **128 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 2, 4, 8, 16



AVX
Vector size: **256 bit**
Data types:
- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 4, 8, 16, 32

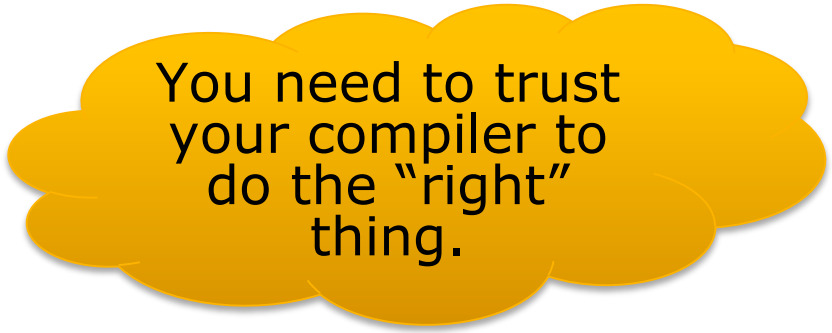
Intel® MIC
Vector size: **512 bit**
Data types:
- 32 bit integer
- 32 and 64 bit float

VL: 8, 16

Illustrations: Xi, Yi & results 32 bit integer

In a Time before OpenMP 4.0…

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



- **SIMD parallelism needed vendor-specific extensions**
  - → Programming models (e.g. C Array Notations)
  - → Compiler pragmas (e.g. #pragma vector)
  - → Low-level constructs

SIMD Loop Example (More Complex)

```
float simd(unsigned offset, unsigned size, float *a) {
  int i;
  int sum = 0;
  float *ptr = a;
#pragma omp simd safelen(4) reduction(+:sum) linear(ptr:1)
  for (i = 0; i < size - offset; i++) {
      a[i + offset] = *ptr; // offset = 4
      ptr += offset / 4; // Always 1 in our example
      if(a[i] > 0.0)
          sum += a[i];
      else
          sum += -1.0;
  }
  return sum;
}
```

SIMD Loop Clauses

- safelen(*length*)
  Maximum number of iterations that can run concurrently without breaking a dependence

- linear(*list[:linear-step]*)
  The variable 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

- private(*list*)
- firstprivate(*list*)
- reduction(*operator:list*)
- collapse(*n*)

} same semantics as in OpenMP 3.1

OpenMP

Well – these codes were all pretty simple. What about something more irregular?

Tasking

Sudoko for Lazy Computer Scientists

OpenMP

- 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) Find an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:

Delete number,

Insert next number

(4 b) If valid:

Go to next field

The OpenMP Task Construct

C/C++

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

Fortran

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

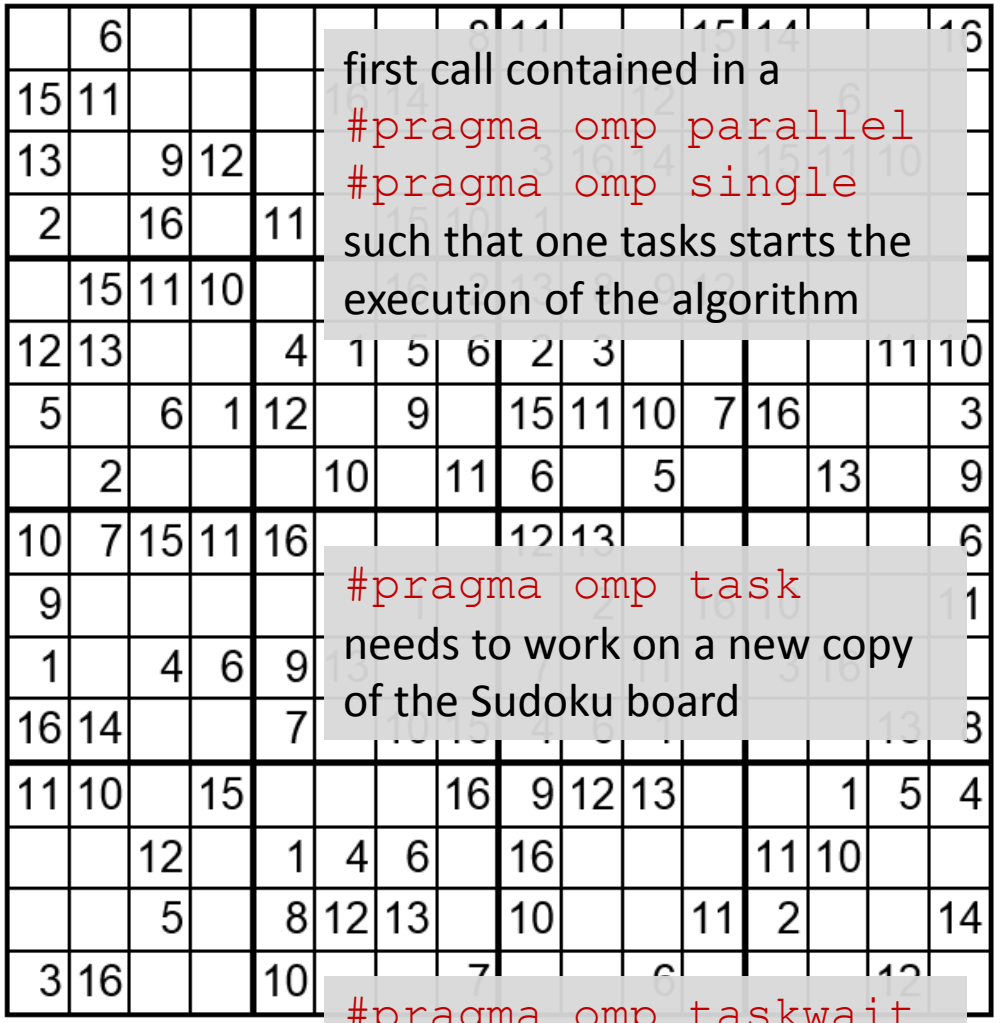
- **Each encountering thread/task creates a new task**
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another task directive
    - → Into a Worksharing construct
- **Data scoping clauses:**
  - → `shared`(*list*)
  - → `private`(*list*) `firstprivate`(*list*)
  - → `default`(*shared*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

OpenMP Task 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 an a task can only be suspended at a suspend point (creation, finish, `taskwait`, `barrier`)
  - → If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all Tasks tied to the thread
- **Task barrier: `taskwait`**
  - → Encountering task suspends until child tasks are complete
    - → Only direct childs, not descendants!
- **OpenMP `barrier` (implicit or explicit)**
  - → All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

Parallel Brute-force Sudoku (1/3)

- This parallel algorithm finds all valid solutions



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) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

(4 b) If valid:
Go to next field

Parallel Brute-force Sudoku (2/3)

- **Create an OpenMP Parallel Region 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 (you‘ll like it or you will not)**

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

Parallel Brute-force Sudoku (3/3)

OpenMP

- **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` needs to work on a new copy of the Sudoku board

`#pragma omp taskwait` wait for all child tasks

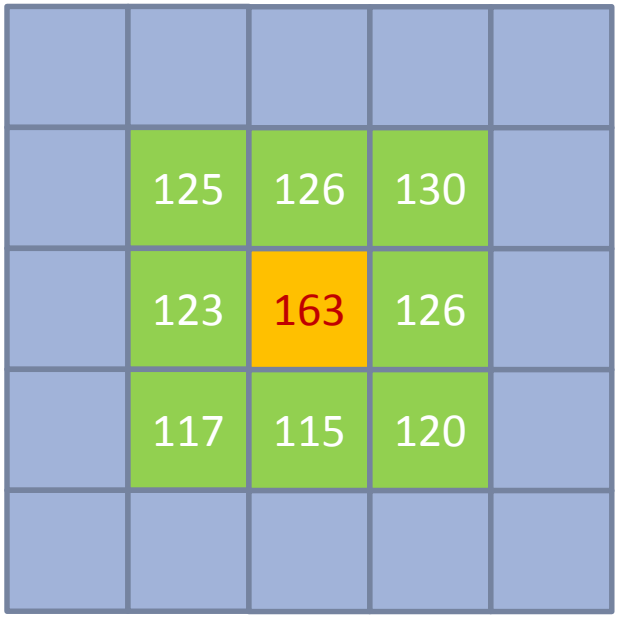
OpenMP

Give me some more power!

Accelerators / Coprocessors

Median Filter

- **Example: Median filter to soften colors in an image**



1. Get color values of neighboring pixels
   125, 126, 130, 123, 163, 126, 117, 115, 120
2. Sort color values (ascending)
   115, 117, 120, 123, 125, 126, 126, 130, 163
3. Choose new color value (median):
   115, 117, 120, 123, 125, 126, 126, 130, 163
4. Update color value

Parallel Median Filter in C/C++

```
void MedianFilter(unsigned* inputArray, unsigned* outputArray,
                  unsigned arrayWidth, unsigned arrayHeight) {
    memset(outputArray, 0, arrayWidth * (arrayHeight+4));

#pragma omp parallel for shared(inputArray,outputArray)
    for(unsigned y = 0; y < arrayHeight; y++) { // rows
        int iOffset = (y+2) * arrayWidth;
        int iPrev = iOffset - arrayWidth;
        int iNext = iOffset + arrayWidth;
        for(unsigned  x = 0; x < arrayWidth; x++) {  // columns
            unsigned uiRGBA[9];
            uiRGBA[0] = inputArray[iPrev + x - 1];
            uiRGBA[1] = inputArray[iPrev + x];
            uiRGBA[2] = inputArray[iPrev + x + 1];
            uiRGBA[3] = inputArray[iOffset + x - 1];
            uiRGBA[4] = inputArray[iOffset + x];
            // ...
```

Parallel Median Filter in C/C++

```
// ...

// bitonic sorting
unsigned uiMin = c4min(uiRGBA[0], uiRGBA[1]);
unsigned uiMax = c4max(uiRGBA[0], uiRGBA[1]);
uiRGBA[0] = uiMin;
uiRGBA[1] = uiMax;
uiMin = c4min(uiRGBA[3], uiRGBA[2]);
uiMax = c4max(uiRGBA[3], uiRGBA[2]);
uiRGBA[3] = uiMin;
uiRGBA[2] = uiMax;
uiMin = c4min(uiRGBA[2], uiRGBA[0]);
uiMax = c4max(uiRGBA[2], uiRGBA[0]);
// ...
```

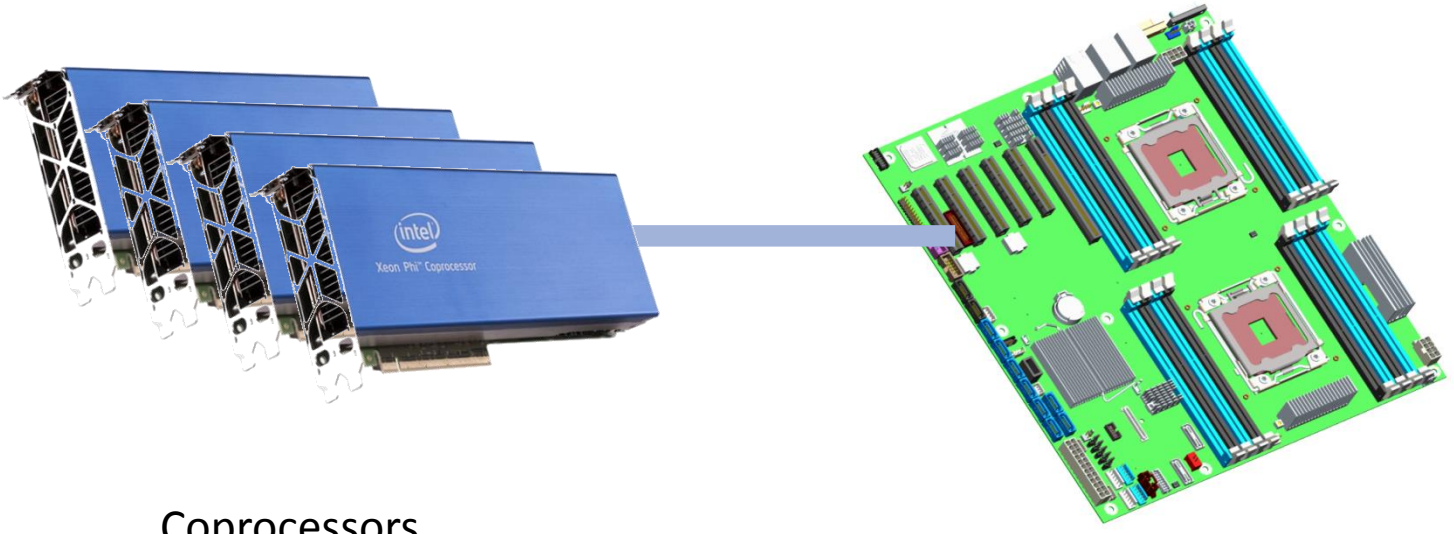
Parallel Median Filter in C/C++

```
            // ...

            uiRGBA[0] = uiMin;
            uiRGBA[8] = uiMax;
            uiRGBA[4] = c4max(uiRGBA[0], uiRGBA[4]);
            uiRGBA[5] = c4max(uiRGBA[1], uiRGBA[5]);
            uiRGBA[6] = c4max(uiRGBA[2], uiRGBA[6]);
            uiRGBA[7] = c4max(uiRGBA[3], uiRGBA[7]);
            uiRGBA[4] = c4min(uiRGBA[4], uiRGBA[6]);
            uiRGBA[5] = c4min(uiRGBA[5], uiRGBA[7]);

            // update pixel
            outputArray[(y+2) * arrayWidth + x] =
                c4min(uiRGBA[4], uiRGBA[5]);
        } // end loop (columns)
    } // end loop (rows)
}
```

Accelerator / Coprocessor Support

- **OpenMP 4.0 introduces support for accelerator/coprocessor devices**

- **Device model:**
  - → One host (with traditional OpenMP threads)
  - → Multiple accelerators/coprocessor (of the same kind)



Coprocessors

Host

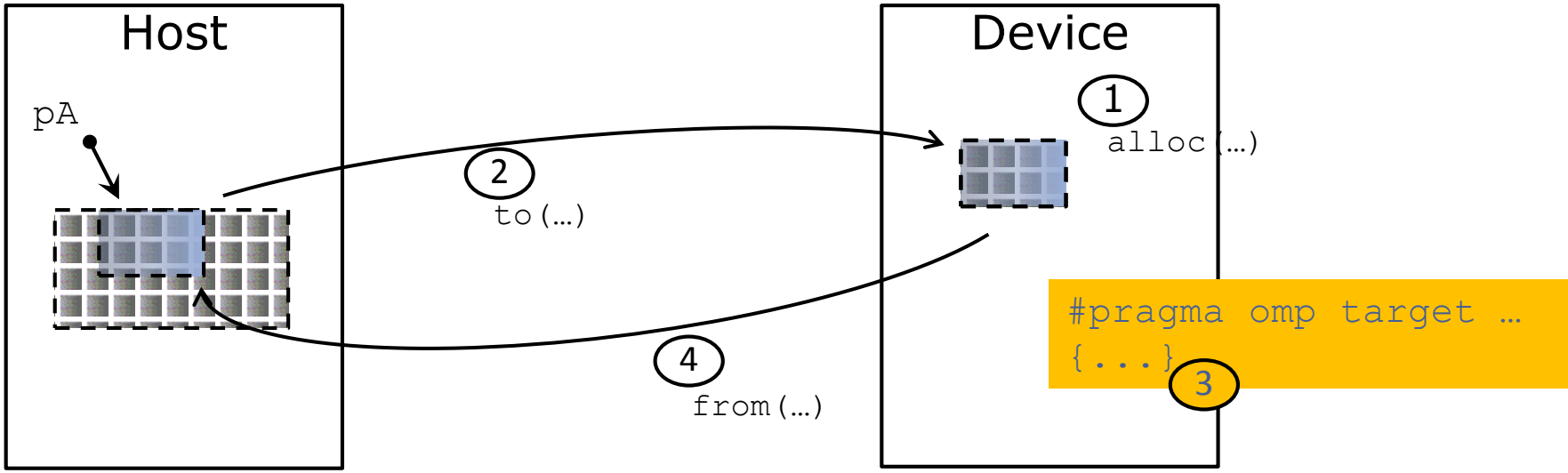
OpenMP Version

```
void MedianFilter(unsigned* inputArray, unsigned* outputArray,
                  unsigned arrayWidth, unsigned arrayHeight) {
    memset(outputArray, 0, arrayWidth * (arrayHeight+4));

#pragma omp target \
    map(in:inputArray[0:arrayWidth * (arrayHeight+4)])
    map(out:outputArray[0:arrayWidth * (arrayHeight+4)])
#pragma omp parallel for shared(inputArray,outputArray)
    for(unsigned y = 0; y < arrayHeight; y++) { // rows
        int iOffset = (y+2) * arrayWidth;
        int iPrev = iOffset - arrayWidth;
        int iNext = iOffset + arrayWidth;
        for(unsigned  x = 0; x < arrayWidth; x++) {  // columns
            unsigned uiRGBA[9];
            uiRGBA[0] = inputArray[iPrev + x - 1];
            uiRGBA[1] = inputArray[iPrev + x];
            // ...
```

Avoiding Unnecessary Data Transfers

```
#pragma omp target data device(0) \
     map(alloc:tmp[0:N]) map(to:input[:N]) map(from:result)
  {
#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(+:result)
    for (i=0; i<N; i++)
      result += final_computation(tmp[i], i)
  }
```

Execution Model



- **The `target` construct transfers the control flow to the target device**
  - → The transfer clauses control direction of data flow
  - → Array notation is used to describe array length
- **The `target data` construct creates a *scoped* device data environment**
  - → The transfer clauses control direction of data flow
  - → Device data environment is valid through the lifetime of the `target data` region
- **Use `target update` to request data transfers from within a `target data` region**

Thank you for your attention!

The OpenMP 4.1 TR3 is available at www.openmp.org.

As we have OpenMP syntax reference cards available at this booth and www.openmp.org, we skipped several syntax details.