Advanced OpenMP Tutorial

Christian Terboven
Michael Klemm
Bronis R. de Supinski

Members of the OpenMP Language Committee

Today's Agenda

- OpenMP Overview — 15 minutes
- Techniques to Obtain High Performance with OpenMP: Memory Access — 45 minutes
- Techniques to Obtain High Performance with OpenMP: Vectorization — 30 minutes
- Advanced Language Features — 60 minutes
- OpenMP for Attached Compute Accelerators — 45 minutes
- Future OpenMP Directions — 15 minutes

Advanced OpenMP Tutorial

An Overview of OpenMP

Christian Terboven
RWTH AACHEN UNIVERSITY

Michael Klemm
intel

Bronis R. de Supinski
Lawrence Livermore National Laboratory

Slides in this section are based on material from Ruud van der Pas (Oracle), who could not come to Leipzig this year.

Members of the OpenMP Language Committee

Agenda

- What is OpenMP?

- Execution Model
- Memory Model
- Directives
- Environment Variables

What is OpenMP?

- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Consists of Compiler Directives, Runtime routines and Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

- Version 3.1 has been released July 2011
- Version 4.0 expected to be released in July 2013

OpenMP ARB Members

Advantages of OpenMP

- OpenMP is ideally suited for multicore architectures
  - → Memory and Threading model map well
  - → Leightweight, Mature, Widely Available and Used
- Good performance and scalability
  - → If you do it right ....
- Requires little programming effort
  - → But, ...
- Allows the program to be parallelized incrementally
  - → But, ...

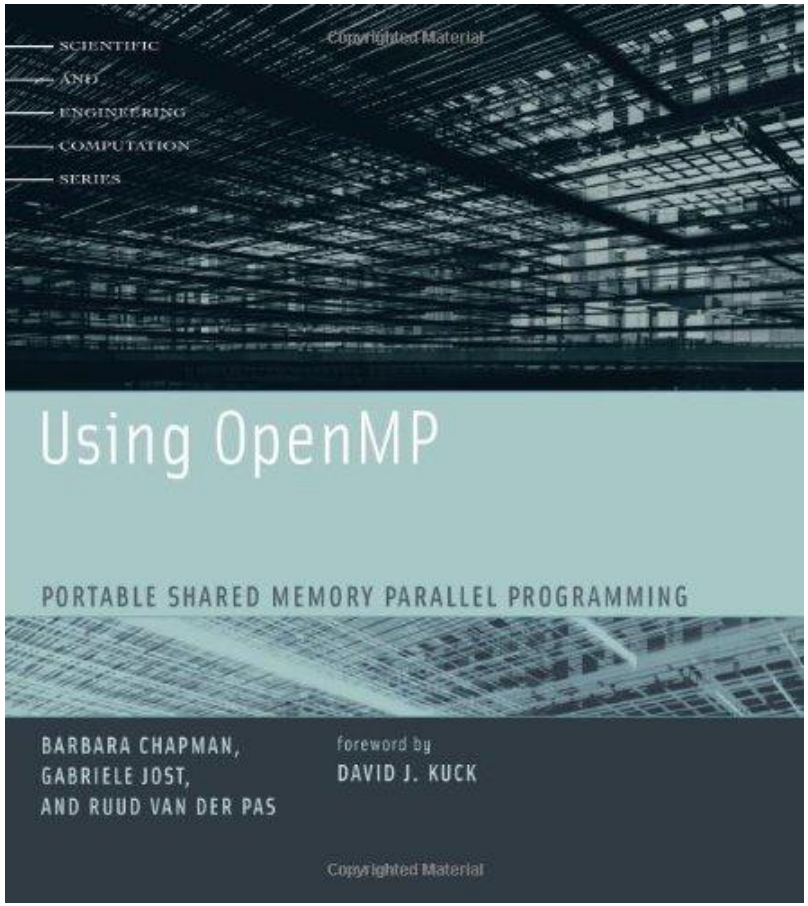
The One Book about OpenMP

- Using OpenMP:
  Portable Shared Memory
  Parallel Programming

  → Chapman, Jost, van der Pas

  → MIT Press, 2008

  → ISBN-10: 0-262-53302-2
  → ISBN-13: 978-0-262-53302-8
  → List price: ~ 35 USD

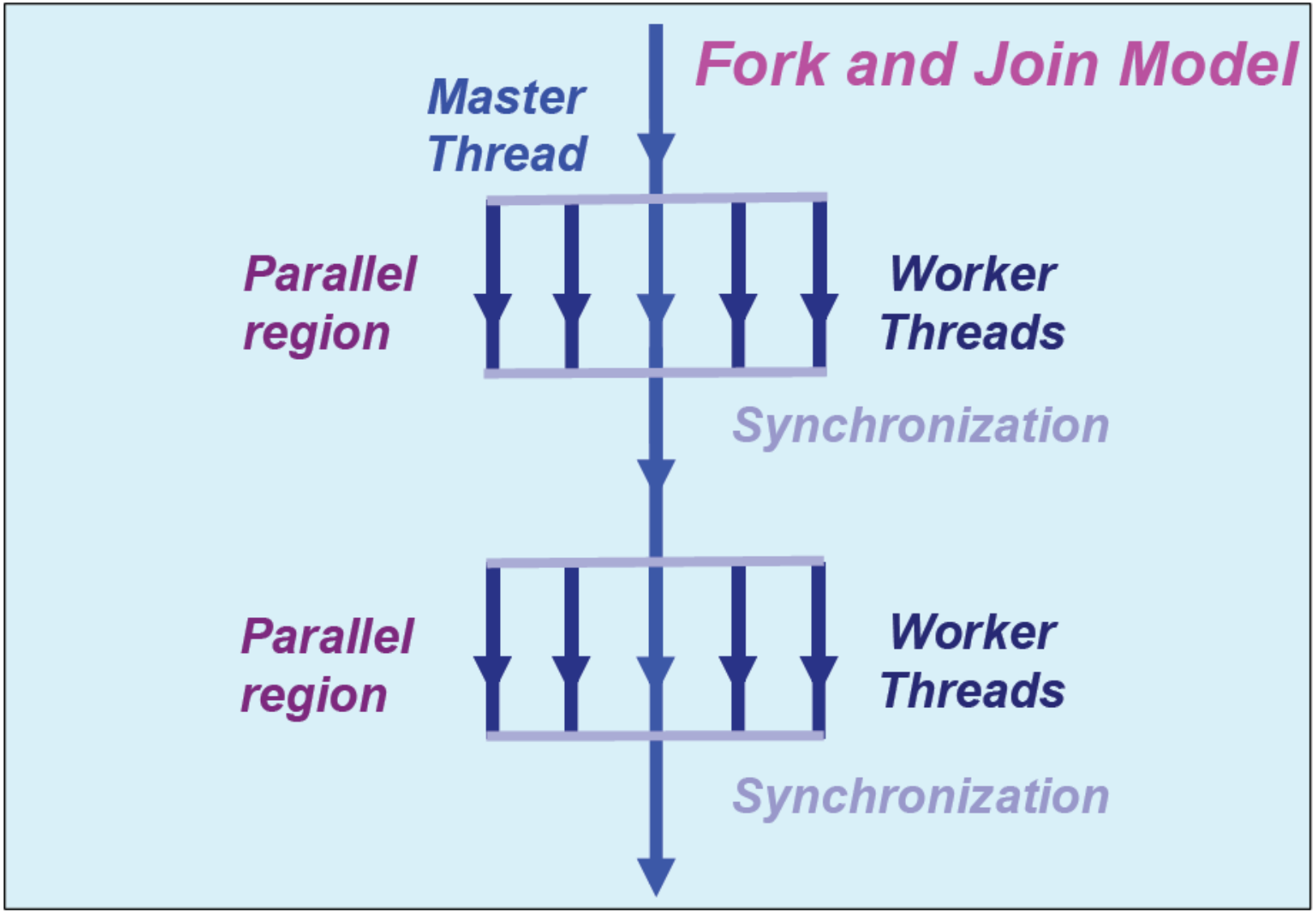


All 41 examples are available on line!

Check out the forum on http://www.openmp.org

Execution Model

The OpenMP Execution Model



The OpenMP Barrier

- Several constructs have an implied barrier
  - →This is another safety net (has implied flush by the way)
- In some cases, the implied barrier can be left out through the “nowait” clause
- This can help fine tuning the application
  - →But you’d better know what you’re doing

- The explicit barrier comes in quite handy then

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

The `nowait` Clause

- To minimize synchronization, some directives support the optional nowait clause
  - → If present, threads do not synchronize/wait at the end of that particular construct
- In C, it is one of the clauses on the pragma
- In Fortran, it is appended at the closing part of the construct

C/C++

```
#pragma omp for nowait
{
    ...
}
```

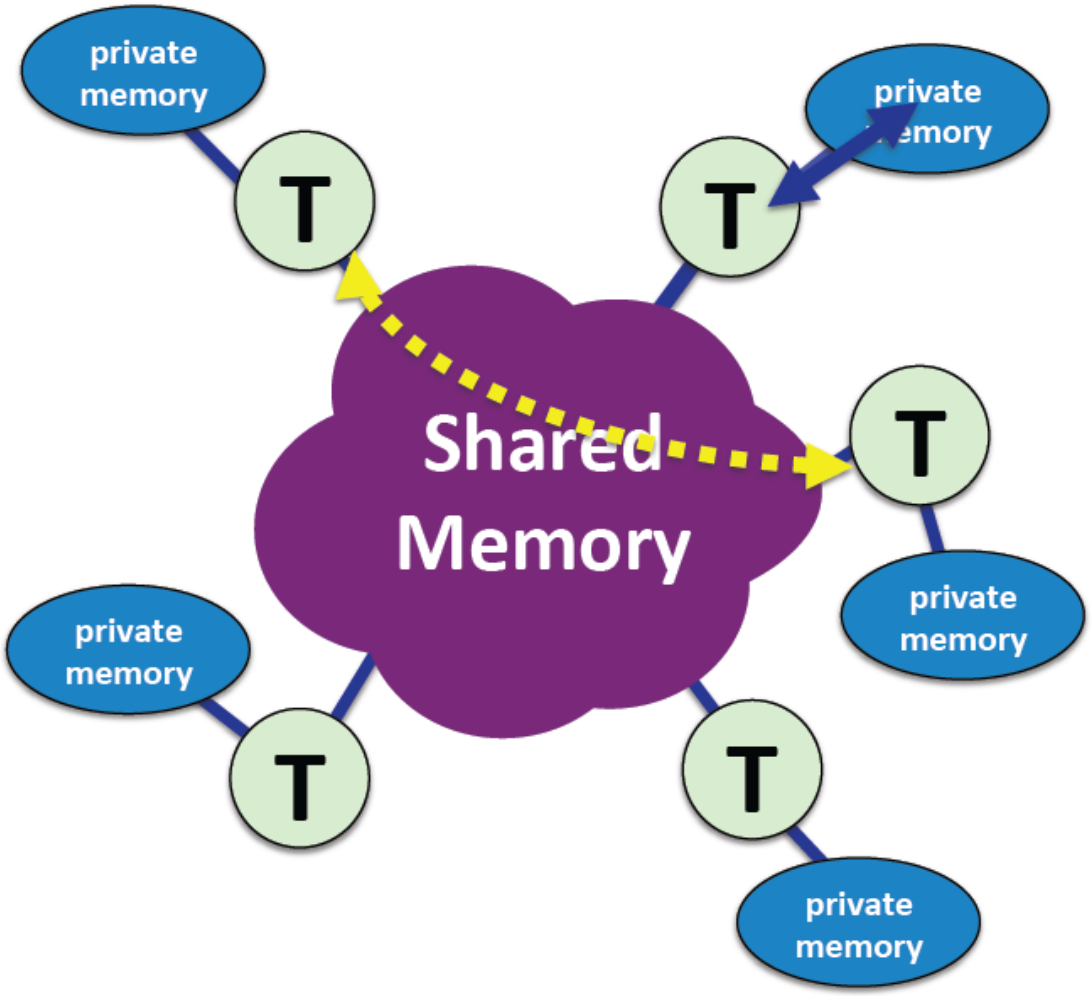
Fortran

```
!$omp do
    ...
    ...
!$omp end do nowait
```

Memory Model

The OpenMP Memory Model (1)

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application

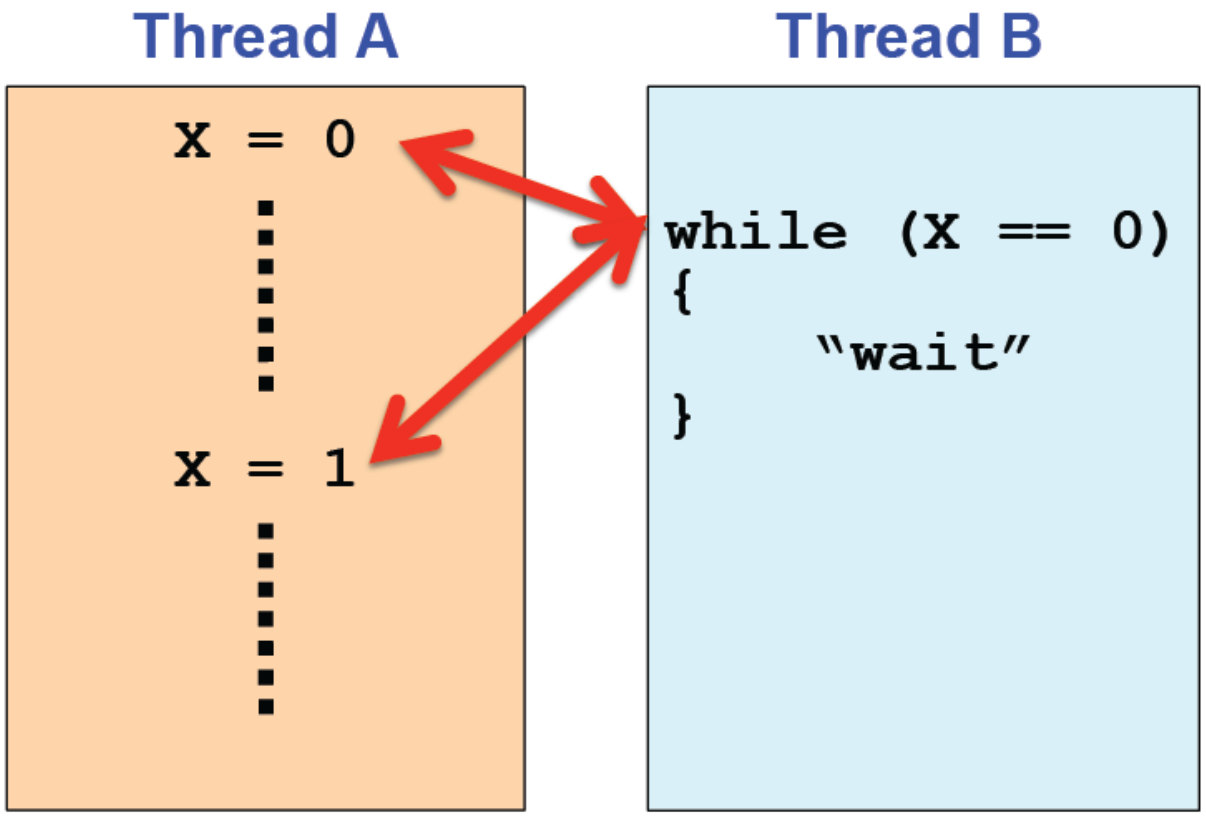


The OpenMP Memory Model (2)

- Need to get this right
  - → Part of the learning curve
- Private data is undefined on entry and exit
  - → Can use firstprivate and lastprivate to address this
- Each thread has its own temporary view on the data
  - → Applicable to shared data only
  - → Means different threads may temporarily not see the same value for the same variable ...
- Let me illustrate the problem we have here…

The `flush` Directive (1)

- If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)



The `flush` Directive (2)

- Example of the Flush Directive, source taken from „Using OpenMP“ pipeline code example

```
void wait_read(int i)
{

   #pragma omp flush

   while ( execution_state[i] != READ_FINISHED )
   {

      system("sleep 1");

      #pragma omp flush
   }

} /*-- End of wait_read --*/
```

The `flush` Directive (3)

- Strongly recommended: do **not** use this directive with a list
  - → Could give very subtle interactions with compilers
  - → If you insist on still doing so, be prepared to face the OpenMP language lawyers
- Implied on many constructs
  - → A good thing
  - → This is your safety net
- Really, try to avoid at all, if possible!

Directives

Defining Parallelism in OpenMP

- A parallel region is a block of code executed by all threads in the team

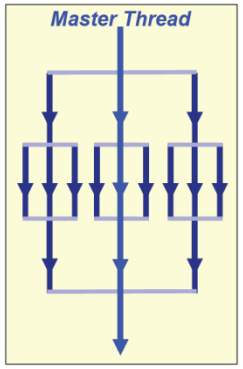
C/C++

```
#pragma omp parallel
{
    ...
    /* parallel *
    ...
} // implied barrier
```

Fortran

```
!$omp parallel
    ...
    ! parallel
    ...
!$omp end parallel
    ! implied barrier
```

- Nested Parallelism



→can be arbitrarily deep

3-way parallel

9-way parallel

3-way parallel

Components of OpenMP

Directives	Runtime Functions	Env. Variables
Array Section Expressions	Number of Threads	Number of Threads
Parallel Region	Thread ID	Scheduling Type
Worksharing Constructs	Dynamic Thread Adjustment	Dynamic Thread Adjustment
SIMD Constructs	Cancellation Status	Thread Affinity Places
Device Constructs	Nested Parallelism	Nested Parallelism
Tasking	Schedlue	Stacksize
Synchronization Constructs	Active Levels	Idle Thread
Cancellation Constructs	Device Selection	Active Levels
Declaration Constructs	Thread Limit	Thread Limit
Memory Flush	Nesting Level	Cancellation
Data-sharing attributes	Ancestor Thread	Default Printout
	Team Size	
	Wallclock Timer	
	Locking	

Red color indicates new addition to OpenMP 4.0

The Worksharing Constructs

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in theam, or non at all
- No implied barrier on entry; implied barrier on exit
- A worksharing construct does not launch any new threads

Some Additional Directives (1)

OpenMP

```
#pragma omp master
{<code-block>}
```

```
!$omp master
     <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
     <code-block>
!$omp end critical [(name)]
```

Very useful to avoid data races

```
#pragma omp atomic
```

```
!$omp atomic
```

Also supports fine tuning controls

Some Additional Directives (2)

OpenMP

```
#pragma omp task
```

```
!$omp task
```

Define a task

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

Wait on completion of child tasks

```
#pragma omp taskyield
```

```
!$omp taskyield
```

Current task can be suspended

Environment Variables

Environment Variables

Name	Possible Values	Most Common Default
OMP_NUM_THREADS	Non-negative Integer	1 or #cores
OMP_SCHEDULE	„schedule [, chunk]“	„static, (N/P)“
OMP_DYNAMIC	{TRUE | FALSE}	TRUE
OMP_NESTED	{TRUE | FALSE}	FALSE
OMP_STACKSIZE	„size [B | K | M | G]“	-
OMP_WAIT_POLICY	{ACTIVE | PASSIVE}	PASSIVE
OMP_MAX_ACTIVE_LEVELS	Non-negative Integer	-
OMP_THREAD_LIMIT	Non-negative Integer	1024
OMP_PROC_BIND	{TRUE | FALSE}	FALSE
OMP_PLACES	Place List	-
OMP_CANCELLATION	{TRUE | FALSE}	FALSE
OMP_DISPLAY_ENV	{TRUE | FALSE}	FALSE
OMP_DEFAULT_DEVICE	Non-negative Integer	-

Advanced OpenMP Tutorial

Performance: Memory Access

Christian Terboven
Michael Klemm
Bronis R. de Supinski

RWTH AACHEN UNIVERSITY
intel
Lawrence Livermore National Laboratory

Members of the OpenMP Language Committee

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 and Performance

- The transparency and ease of use of OpenMP are a mixed blessing
  - → Makes things pretty easy
  - → May mask performance bottlenecks
- In an ideal world, an OpenMP application "just runs well". Unfortunately, this is not always the case…

- 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

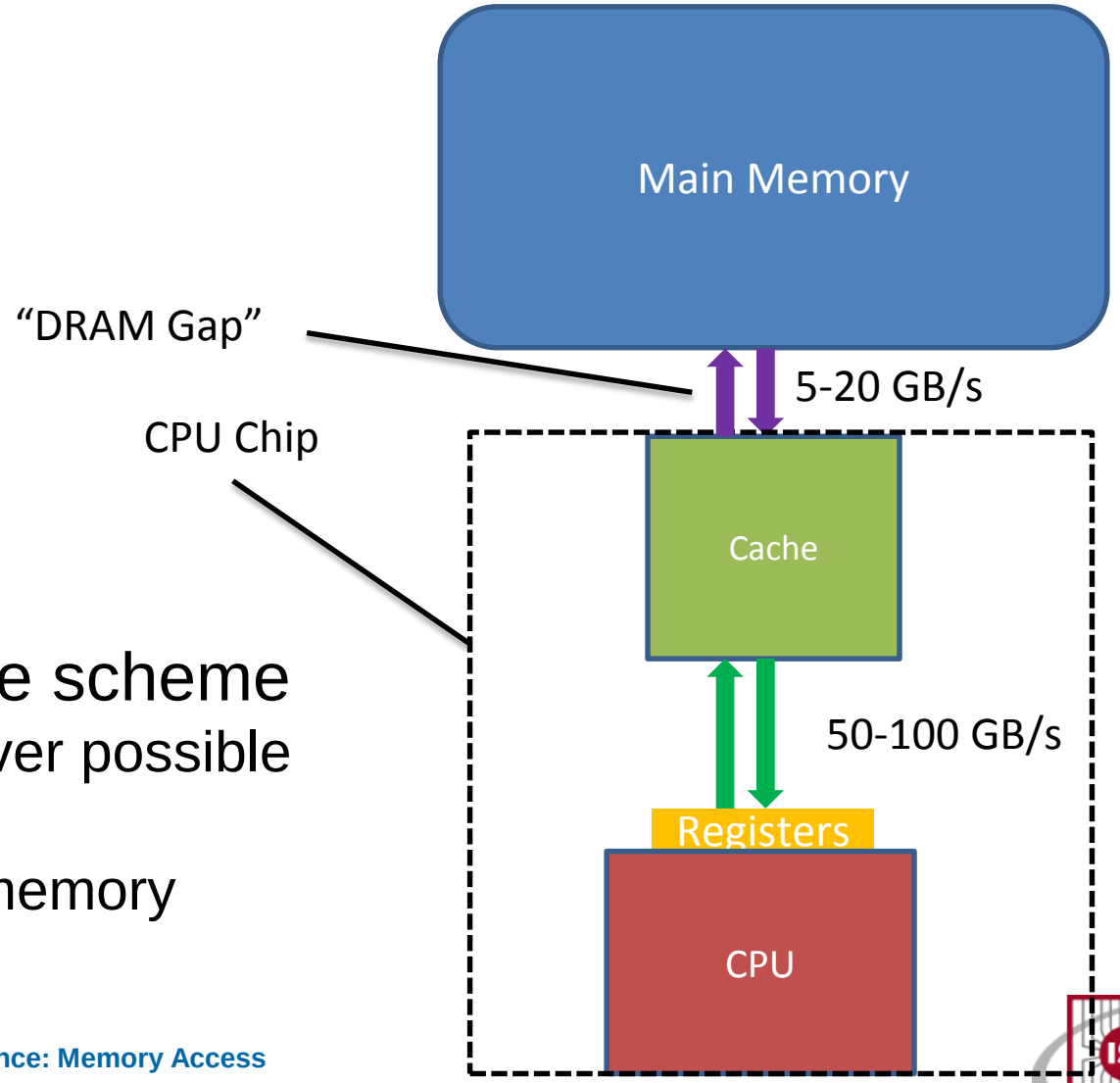
Agenda

- cc-NUMA and False Sharing
- Thread Affinity in OpenMP 4.0
- Example: Bounding Box
- Example: Matrix Vector Multiplication

cc-NUMA and False Sharing

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



Cache Coherence (cc)

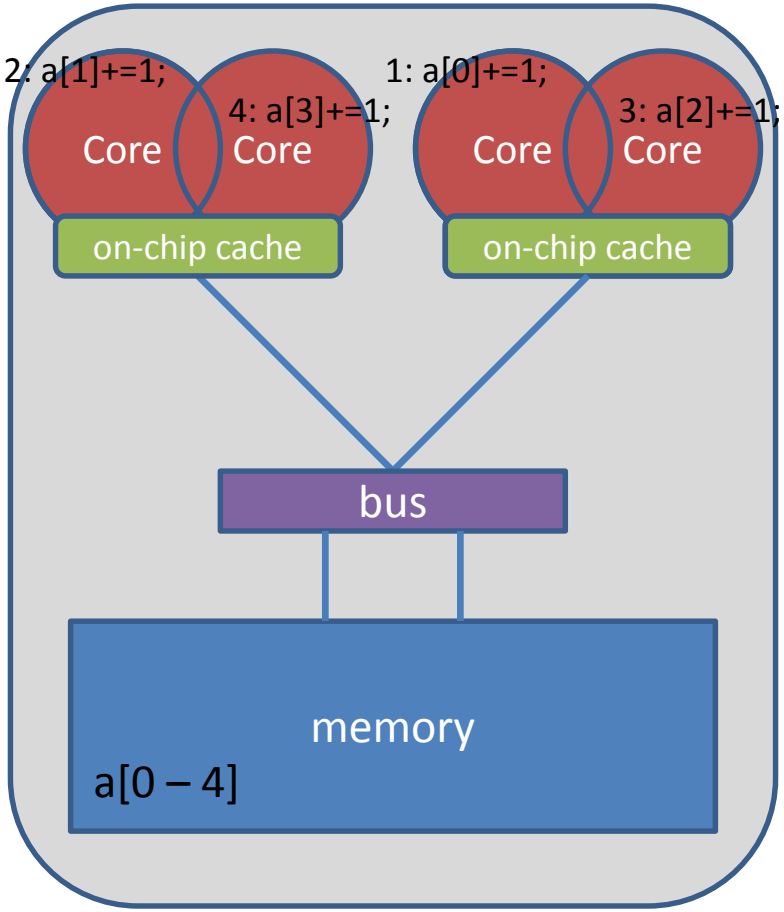
- If there are multiple caches not shared by all cores in the system, the system takes care of the cache coherence.
- Example:

```
int a[some_number]; //shared by all threads
thread 1: a[0] = 23;      thread 2: a[1] = 42;
--- thread + memory synchronization (barrier) ---
thread 1: x = a[1];       thread 2: y = a[0];
```

  - → Elements of array `a` are stored in continuous memory range
  - → Data is loaded into cache in 64 byte blocks (cache line)
  - → Both `a[0]` and `a[1]` are stored in caches of thread 1 and 2
  - → After synchronization point all threads need to have the same view of (shared) main memory
- The system is not able to distinguish between changes within one individual cache line.

False Sharing

- False Sharing: storing data into a shared cache line invalidates the other copies of that line!



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.

False Sharing Indicators

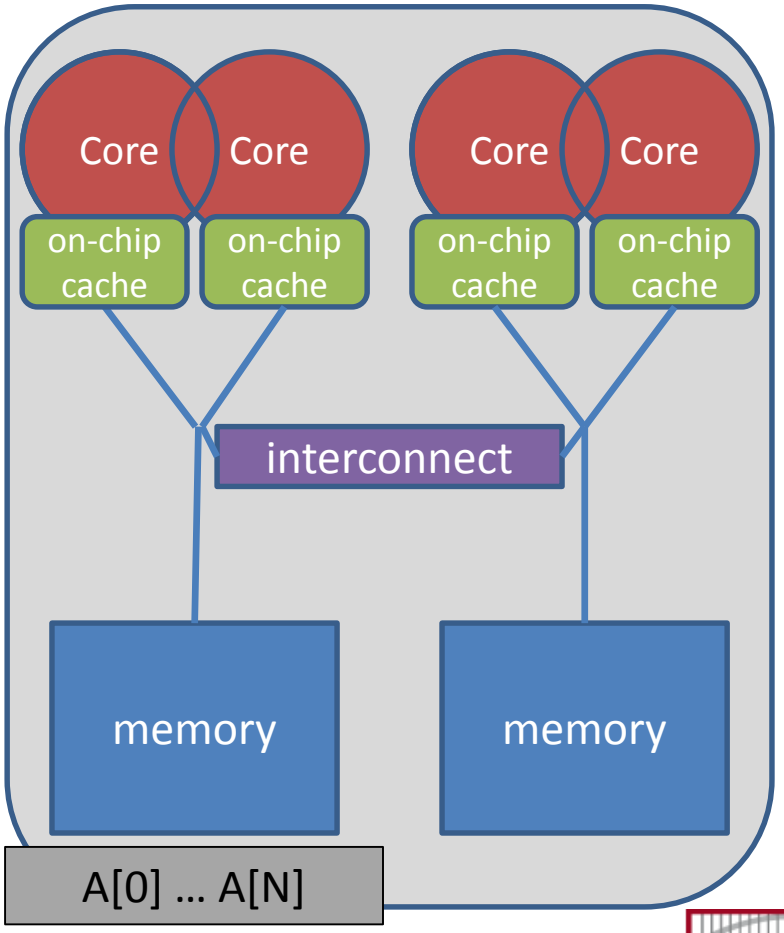
- Be alert, if all of these three conditions are met
  - → Shared data is modified by multiple processors
  - → Multiple threads operate on the same cache line(s)
  - → Update occurs simultaneously and very frequently
- Use local data where possible
- Shared read-only data does not lead to false sharing

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));
```

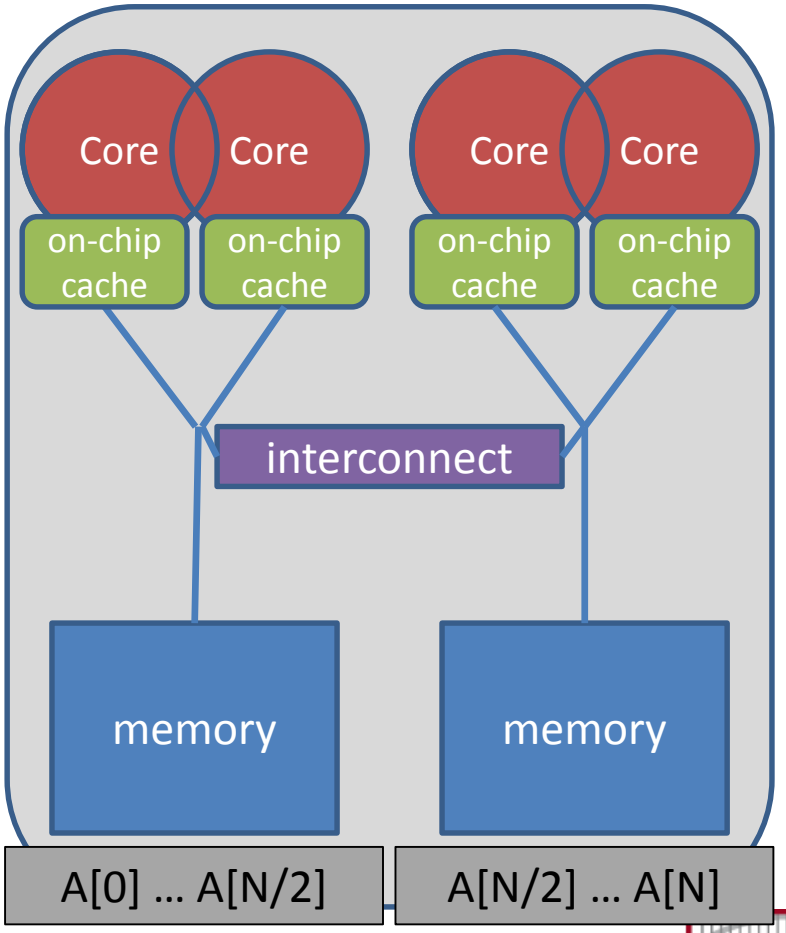
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

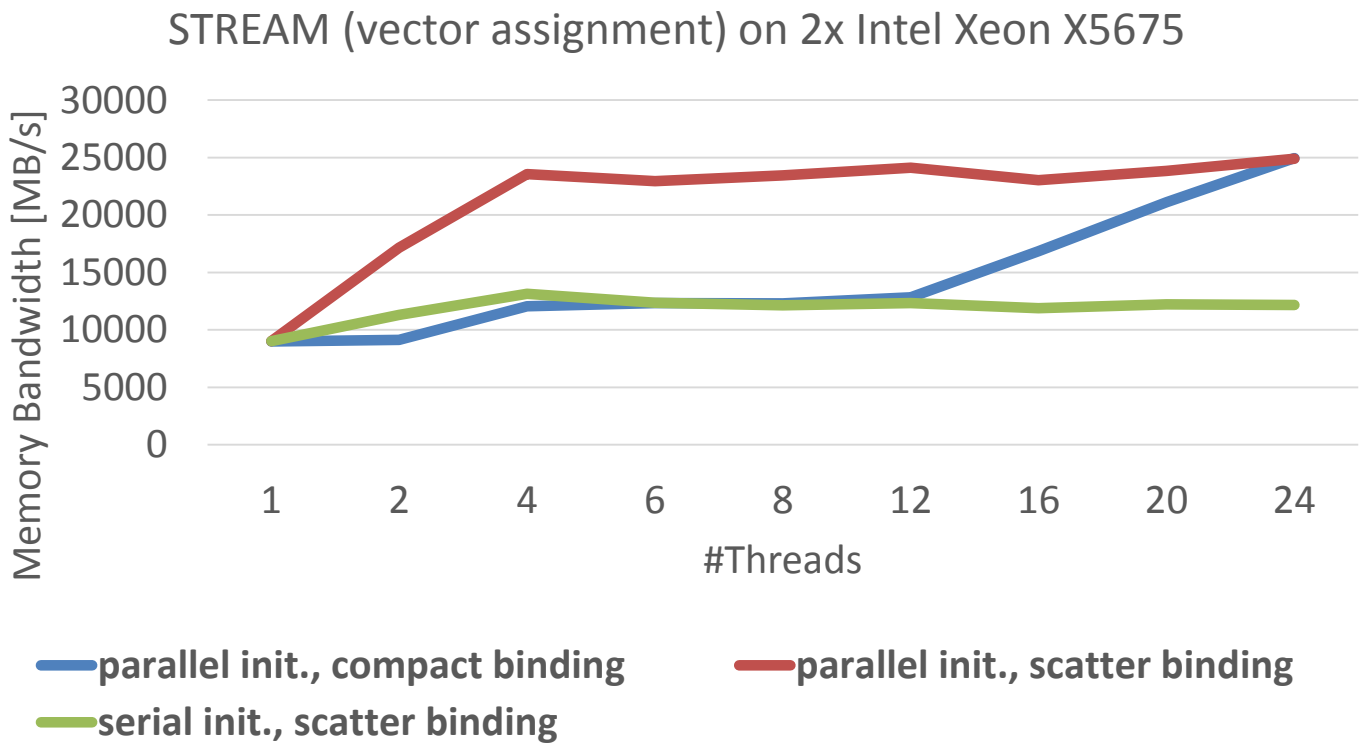
- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```
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;
}
```



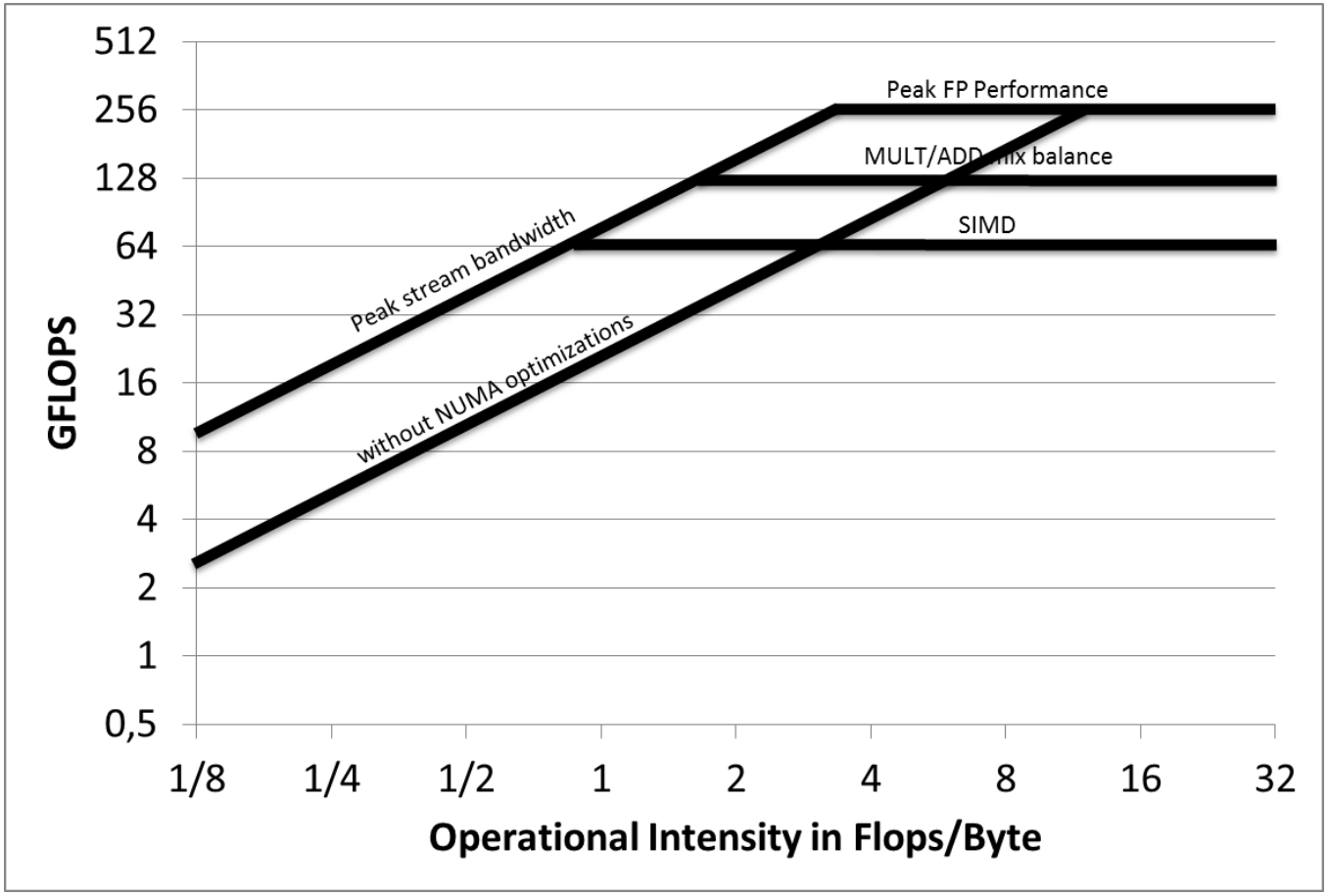
Serial vs. Parallel Initialization

- 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:



Roofline Model

- Peak Performance is only achievable if everything is done right (NUMA, Vectorization, FLOPS, …)!

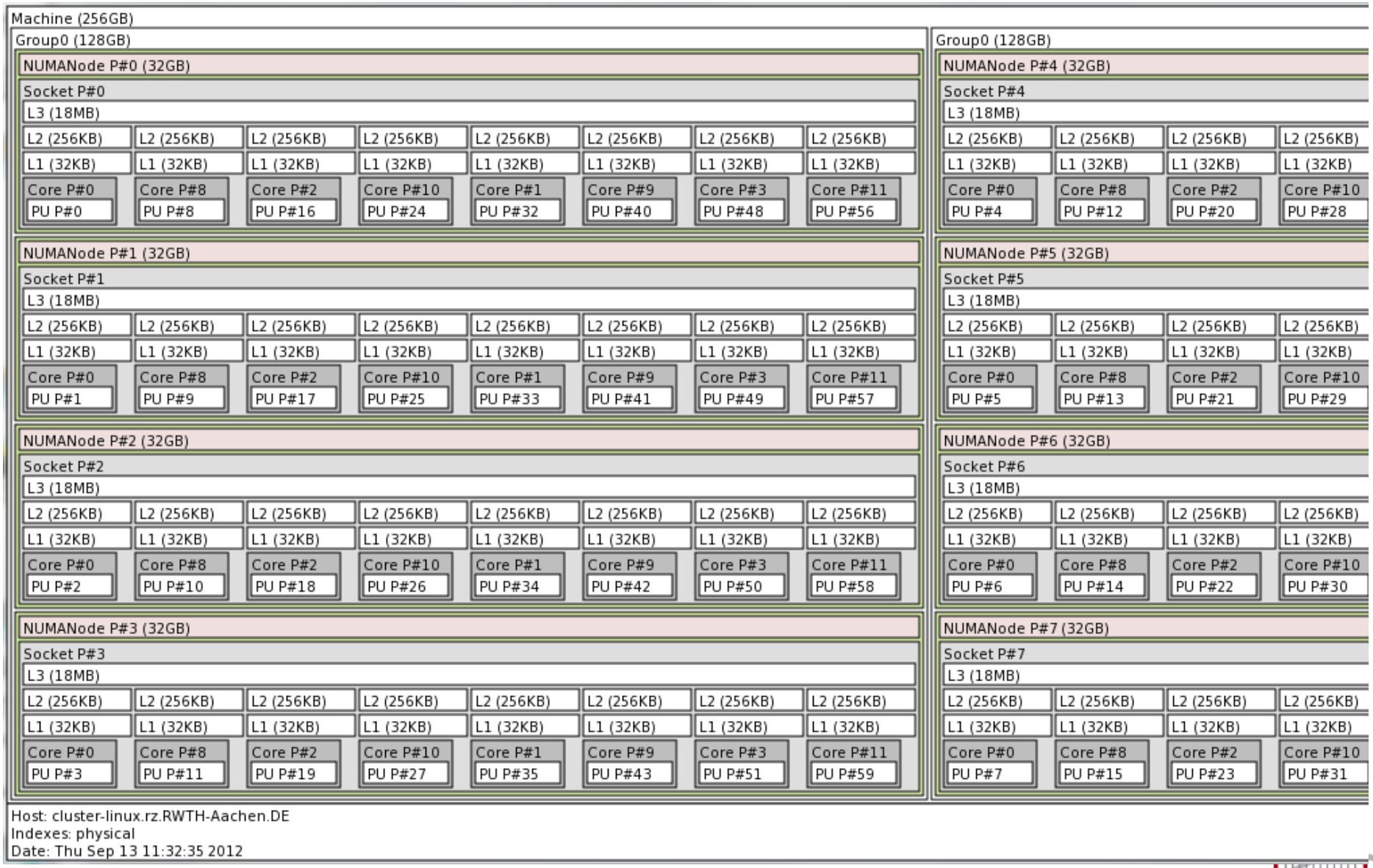


Tools to examine the Hardware Architecture

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
    - Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.
  - hwlocs' `hwloc-ls` tool: http://www.open-mpi.de/projects/hwloc/
    - Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.
  - Reading `/proc/cpuinfo`

Example: hwloc-ls



Thread Affinity in OpenMP 4.0

Deciding for a 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 which 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 (1)

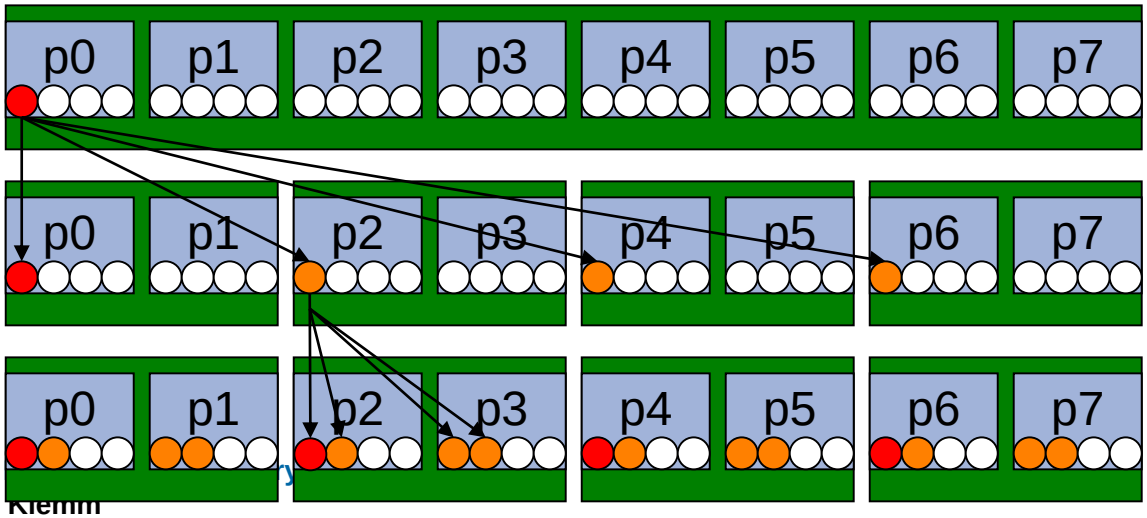
- Define OpenMP Places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user
- Define a set of OpenMP Thread Affinity Policies
  - → SPREAD: spread OpenMP threads evenly among the places
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- Goals
  - → user has a way to specify where to execute OpenMP threads
  - → locality between OpenMP threads / less false sharing / memory bandwidth

OpenMP 4.0: Places + Policies (2)

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

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

- Example
  - →initial
  - →spread 4
  - →compact 4



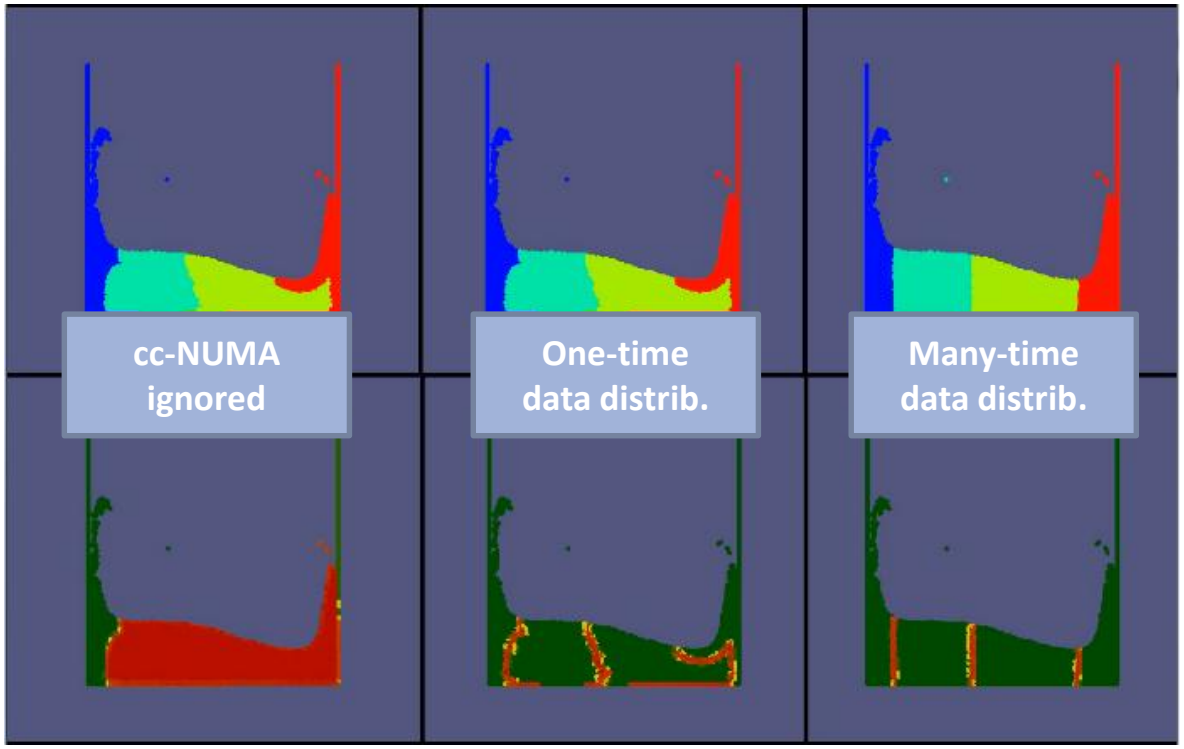
More NUMA Strategies

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?

Example: SPH

- SPH = Smoothed Particle Hydrodynamics, fluid simulation
- Water is represented as a set of (many) particles
- Domain / Grid changes over time …



Every color represents one thread being executed on one socket

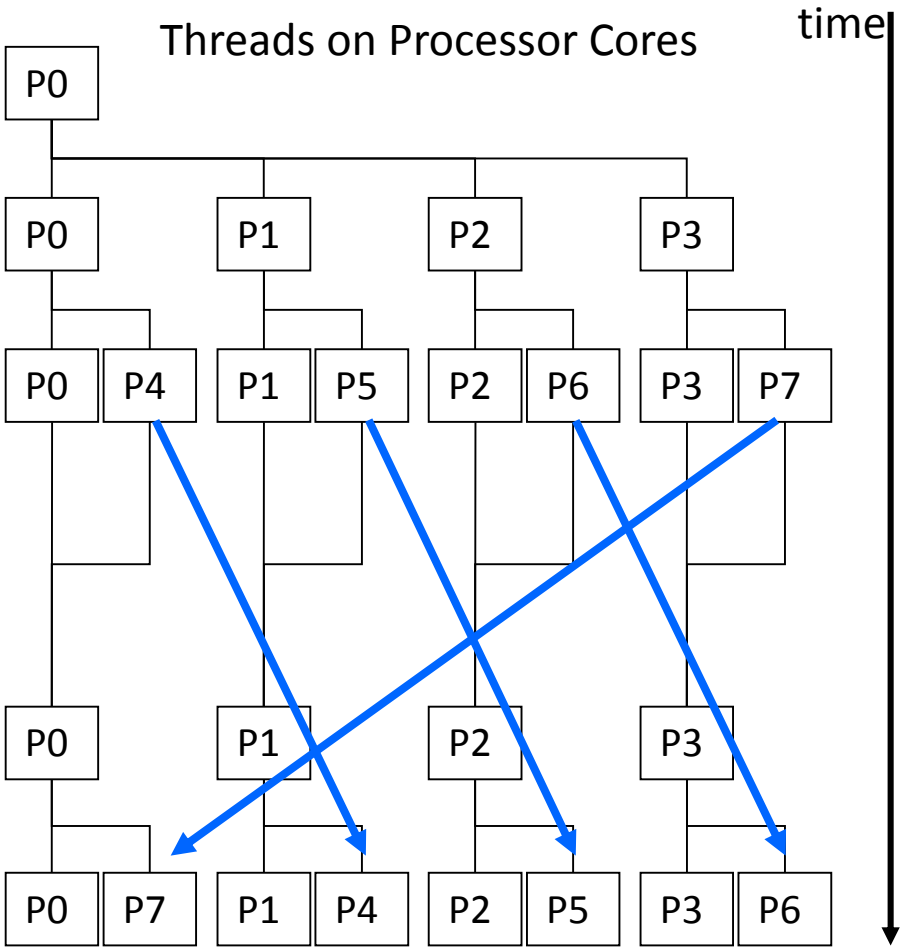
Green = local access
Red = remote access

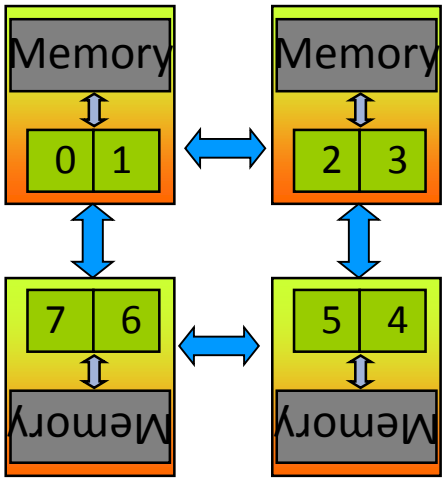
NUMA Strategies: Overview

- First Touch: Modern operating systems (i.e. Linux >= 2.4) decide for a physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU causing 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 where the next „touch“ comes from. Well-supported in Solaris, expensive to implement in Linux.
- Automatic Migration: No support for this in current operating systems.

Nested OpenMP Par. Regions (1)

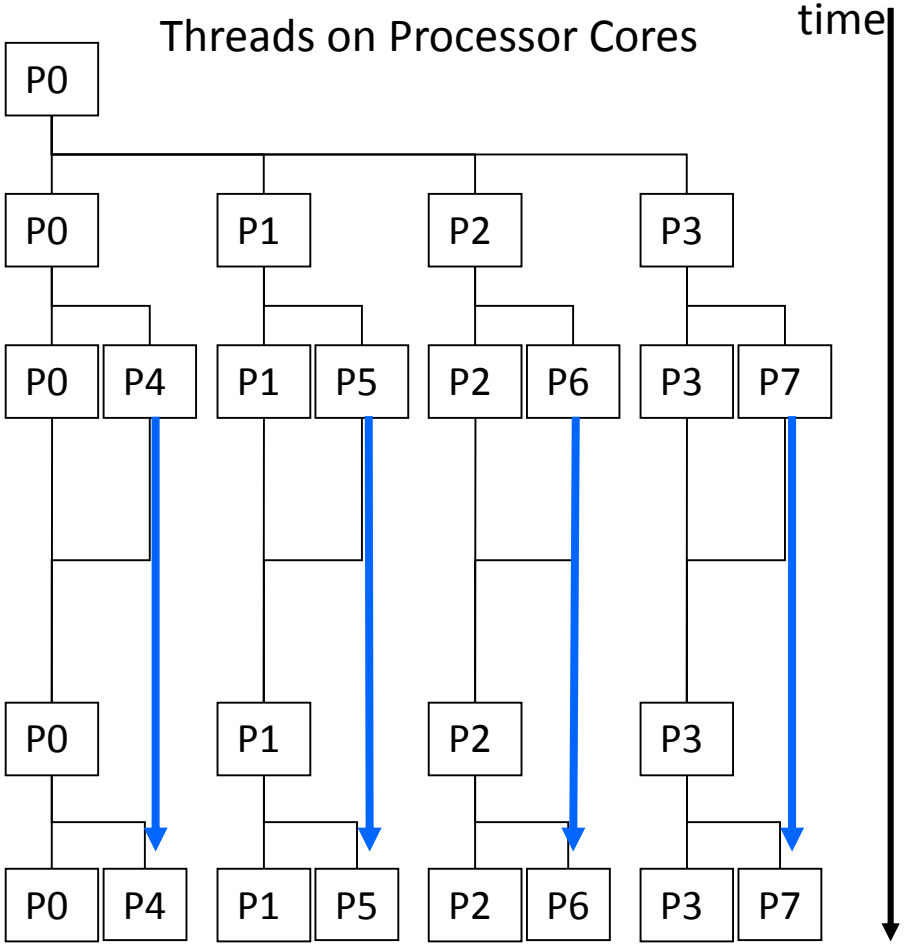
- The OpenMP runtime organizes threads in a pool.





Nested OpenMP Par. Regions (2)

- This is what you want – experimental extension in the Oracle runtime



`SUNW_MP_PROCBIND=true`
Binds threads to cores in the order in which they were created.

`SUNW_MP_THR_AFFINITY=true`
Maintains thread assign ments (on the cost of possible resource waste) .

OpenMP 4.0 supports this via *places*, implementations will be released soon!

Example: Bounding Box

Example: Bounding Box Code

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

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

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

Bounding Box w/ OpenMP: ☹

- The „do not“ approach:
  - → Manual implementation of reduction operation with lb, ub as arrays

```
Point2D *lbs = new Point2D[omp_get_max_threads()], ubs = ...
#pragma omp parallel
{
   lbs[omp_get_thread_num()] = Point2D(RANGE, RANGE); ubs= ...

   /* in the loop: use tid = omp_get_thread_num() again */
   /* see next slide for loop body */

#pragma omp single
   for (int t = 0; t < omp_get_num_threads(); t++) {
      lb.setX(std::min(lb.getX(), lbs[t].getX())); ...
   }
} // end omp parallel
```

Bounding Box w/ OpenMP: ☹

- The „do not“ approach:
  - →Rewrite iterator-loop to for-loop

```
int size = points.size();

#pragma omp for
for (int i = 0; i < size; i++) {
        Point2D &p = points[i];
        // tid == omp_get_thread_num()
        lbs[tid].setX(std::min(lbs[tid].getX(), p.getX()));
        lbs[tid].setY(std::min(lbs[tid].getY(), p.getY()));
        ubs[tid].setX(std::max(ubs[tid].getX(), p.getX()));
        ubs[tid].setY(std::max(ubs[tid].getY(), p.getY()));
} // end omp for
```

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++) {
              ...
```

Bounding Box w/ OpenMP 4.0

- 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:

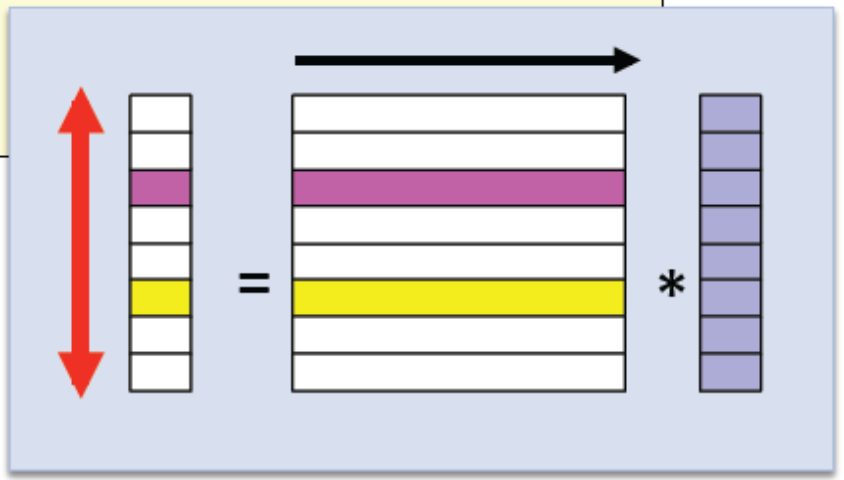
```
std::vector<Point2D, no_init_allocator>
points(NUM_POINTS);
```

Example: Matrix Vector Multiplication

MxV - serial

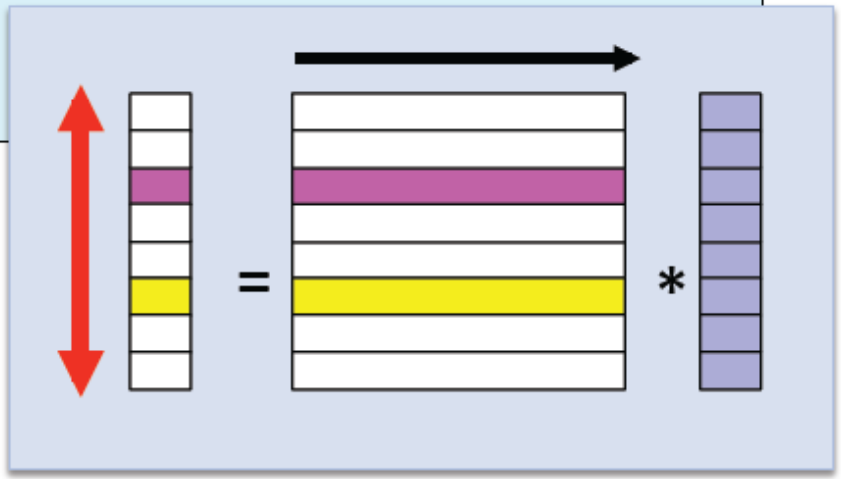
```
void mxv(int m,int n,double *a,double *b[],
         double *c)
{
   for (int i=0; i<m; i++)
   {
      double sum = 0.0;
      for (int j=0; j<n; j++)
         sum += b[i][j]*c[j];
      a[i] = sum;
   }
}
```

parallel loop

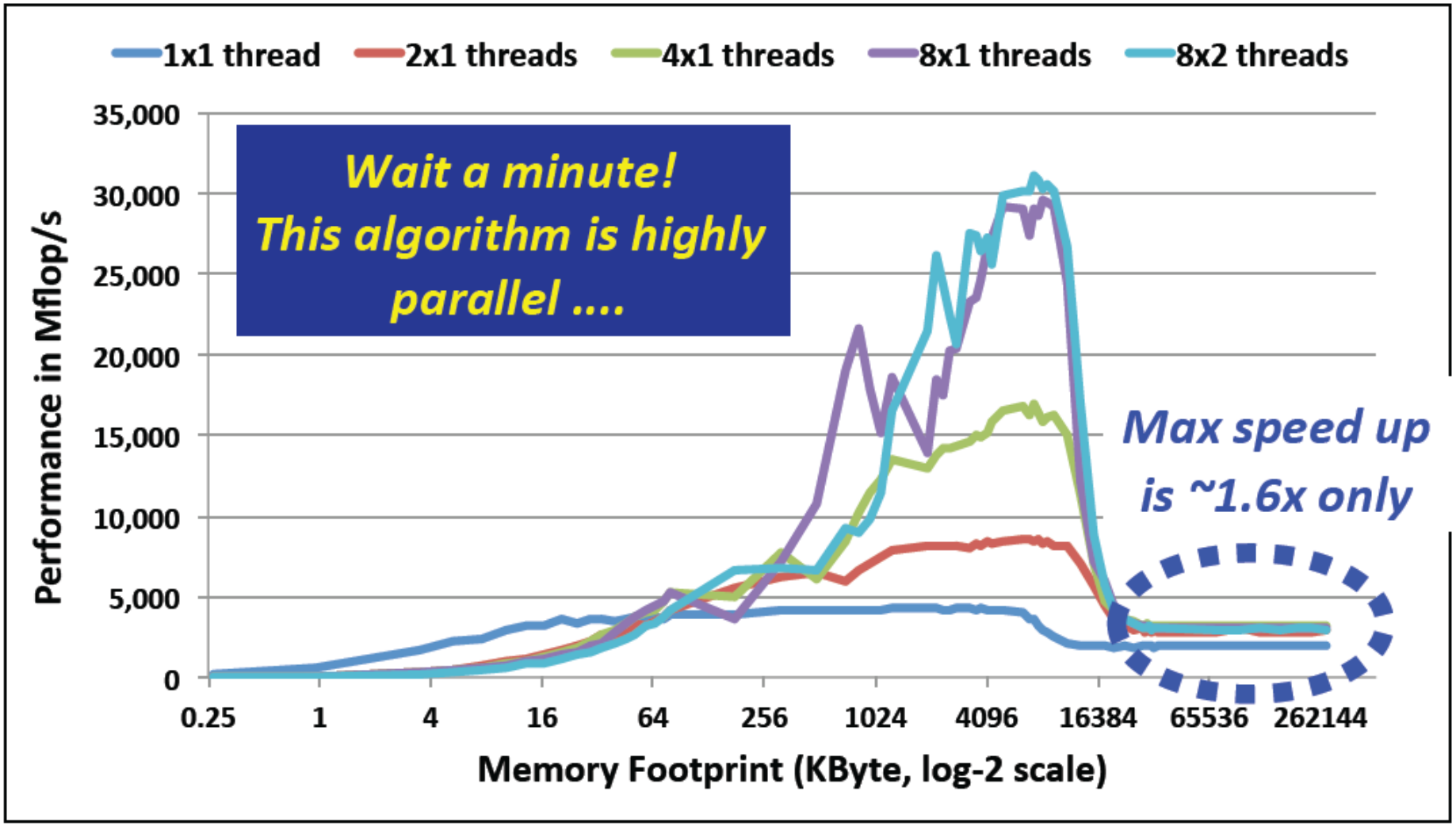


MxV – parallelized with OpenMP

```
#pragma omp parallel for default(none) \
        shared(m,n,a,b,c)
for (int i=0; i<m; i++)
{
  double sum = 0.0;
  for (int j=0; j<n; j++)
    sum += b[i][j]*c[j];
  a[i] = sum;
}
```



Performance on 2x Intel Nehalem

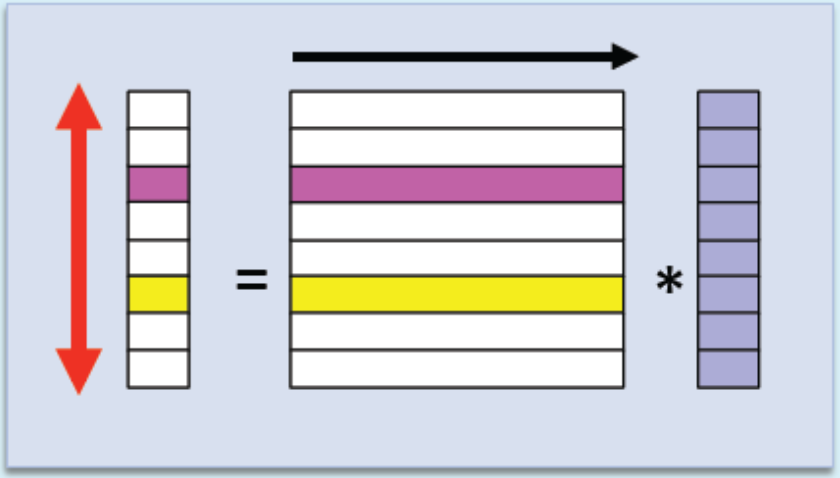


System: Intel X5570 with 2 sockets, 8 cores, 16 threads at 2.93 GHz

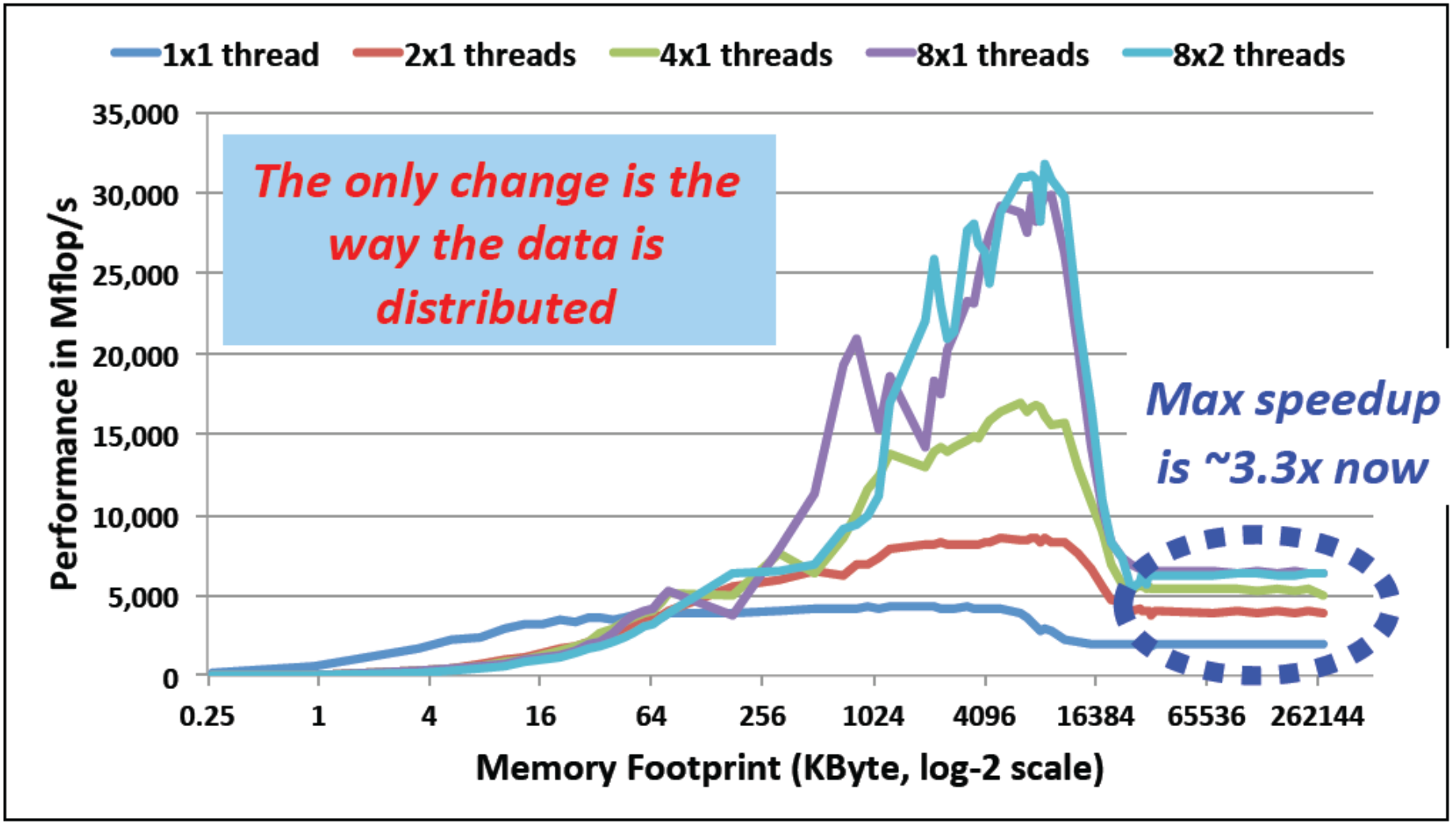
Notation: Number of cores x number of threads within core

Data Initialization Revisited

```
#pragma omp parallel default(none) \
        shared(m,n,a,b,c) private(i,j)
{
#pragma omp for
    for (j=0; j<n; j++)
       c[j] = 1.0;

#pragma omp for
    for (i=0; i<m; i++)
    {
       a[i] = -1957.0;
       for (j=0; j<n; j++)
          b[i][j] = i;
    } /*-- End of omp for --*/

} /*-- End of parallel region --*/
```



Data Placement Matters!



Advanced OpenMP Tutorial

Performance: Vectorization

Christian Terboven
Michael Klemm
Bronis R. de Supinski

RWTH AACHEN UNIVERSITY
intel
Lawrence Livermore National Laboratory

Members of the OpenMP Language Committee

Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

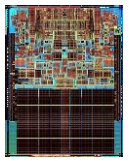
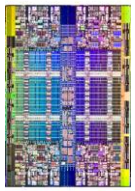
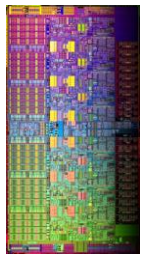
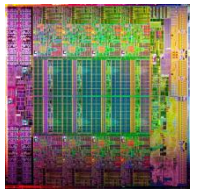
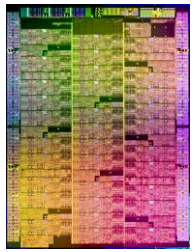
Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

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 Hardware (Intel)

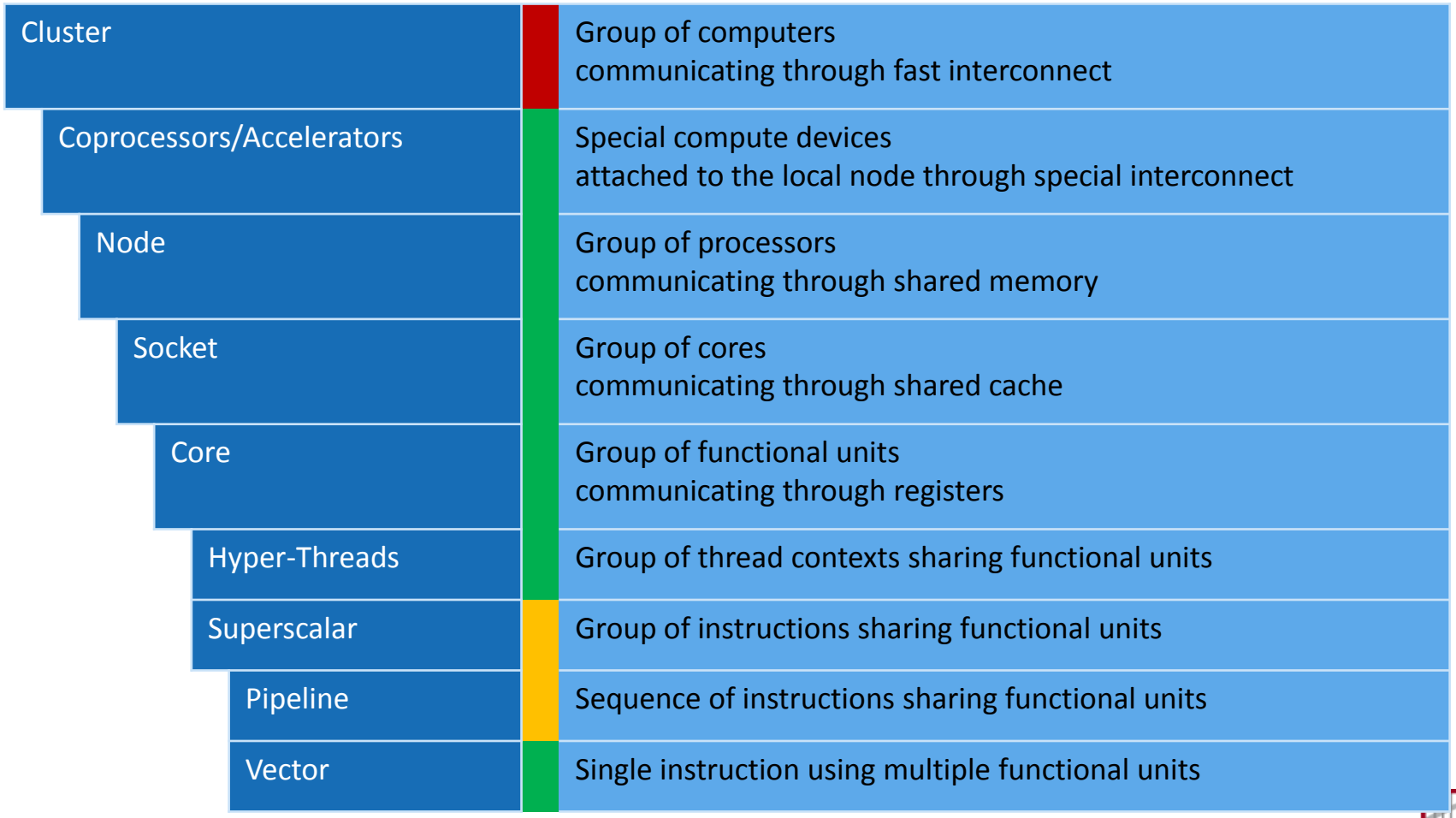







Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600 series	Intel® Xeon Phi™ Co-processor 5110P
Frequency	3.6GHz	3.0GHz	3.2GHz	3.3GHz	2.7GHz	1053MHz
Core(s)	1	2	4	6	8	60
Thread(s)	2	2	8	12	16	240
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

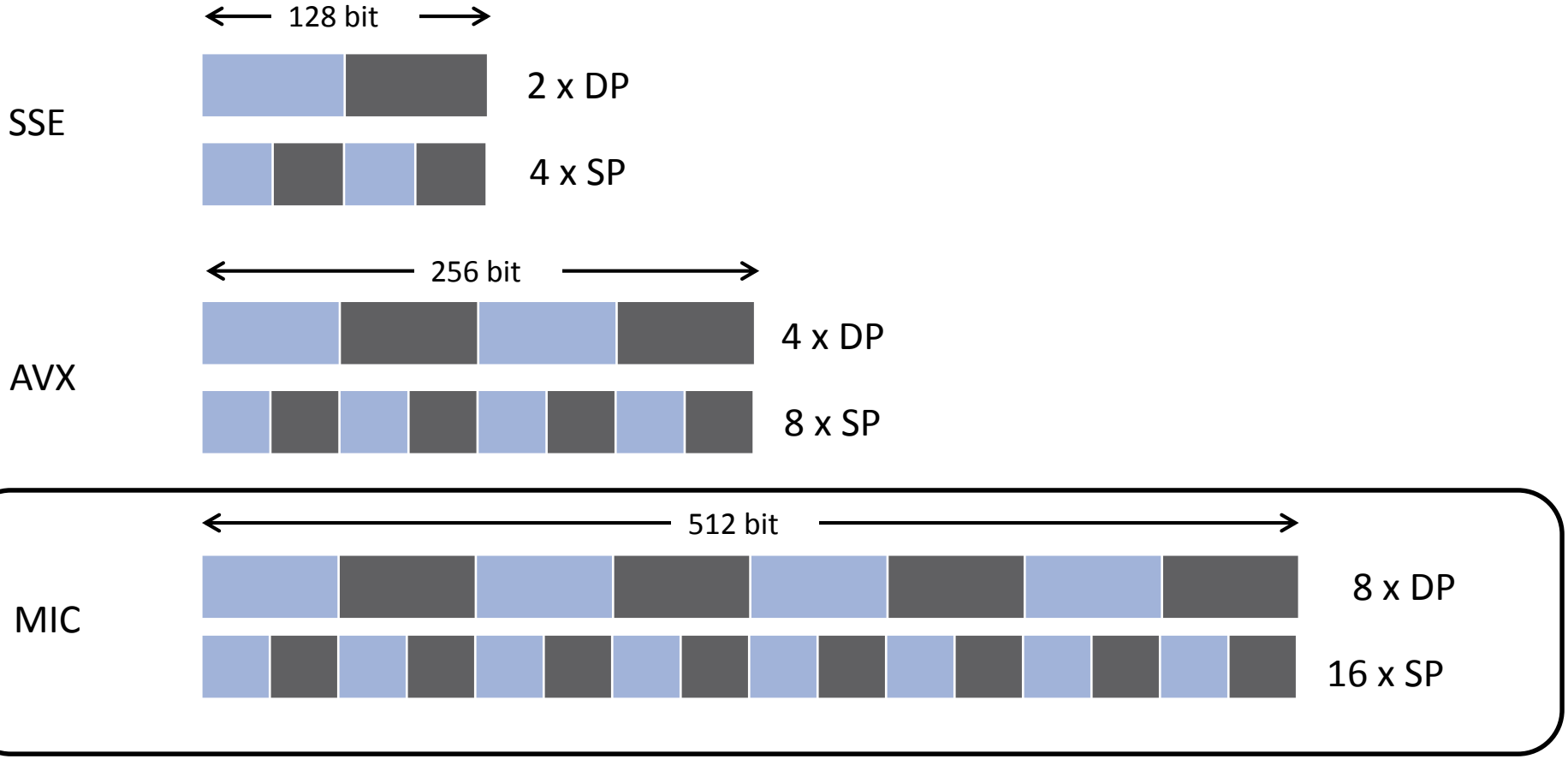
Levels of Parallelism

- OpenMP already supports several levels of parallelism in today's hardware



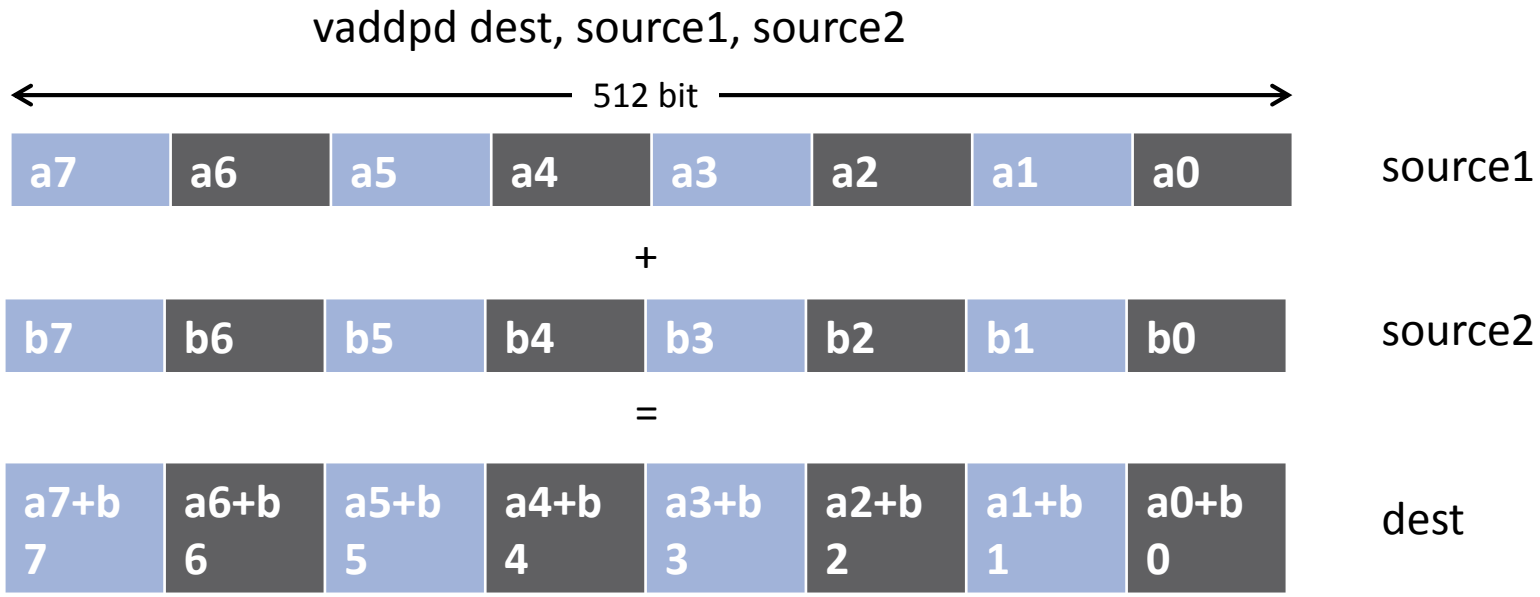
SIMD on Intel® Architecture

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



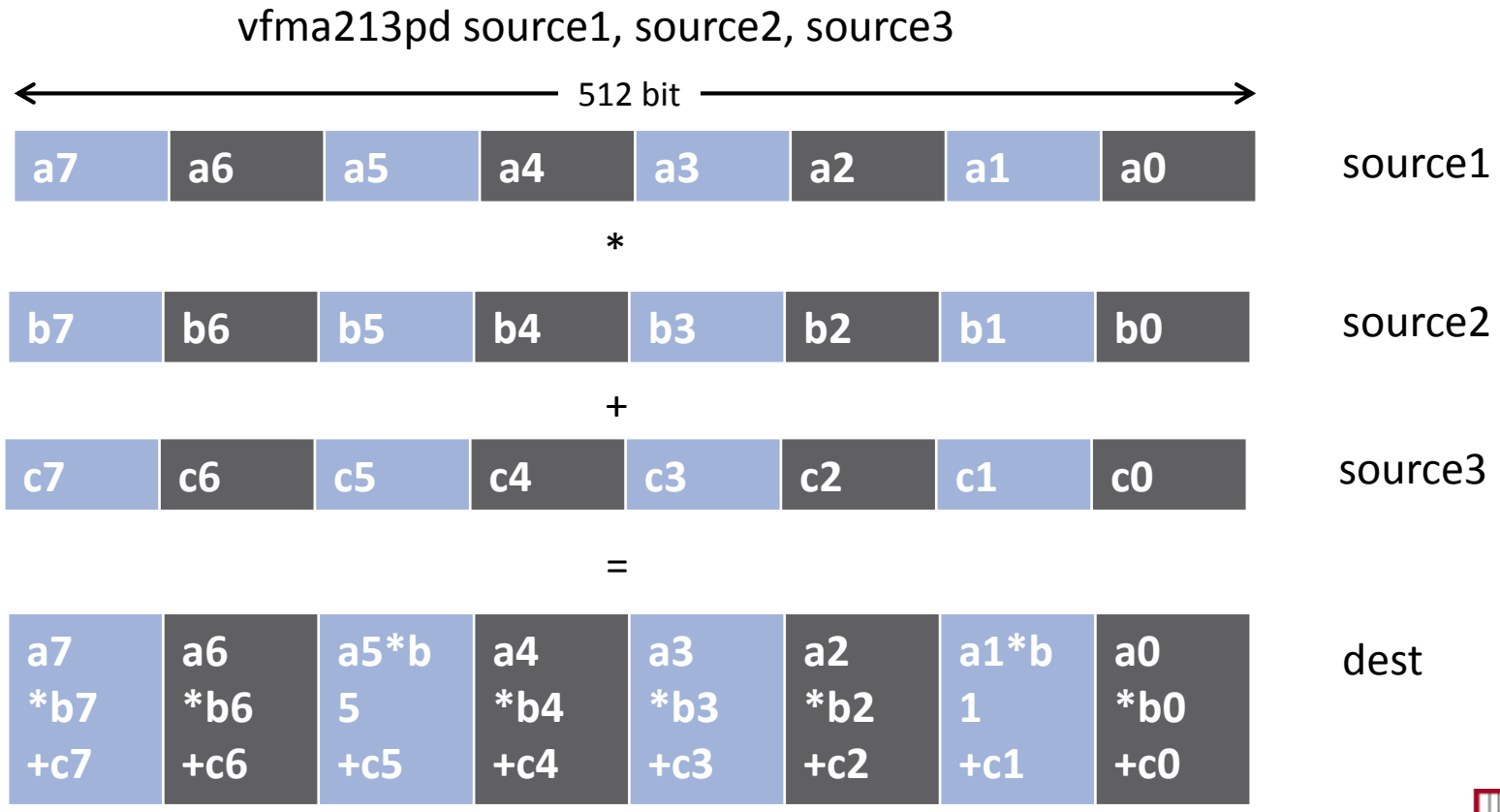
More Powerful SIMD Units

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



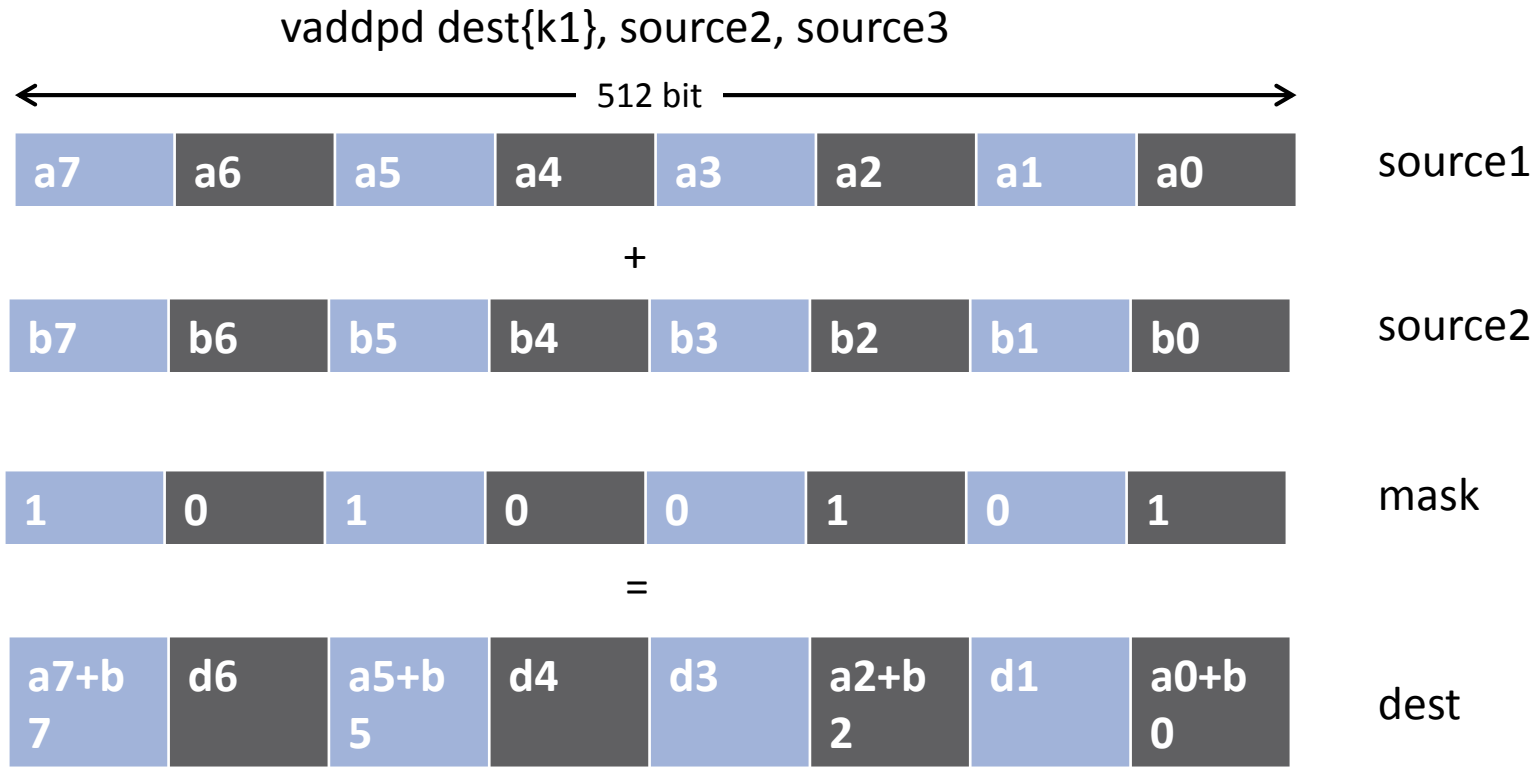
More Powerful SIMD Units

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



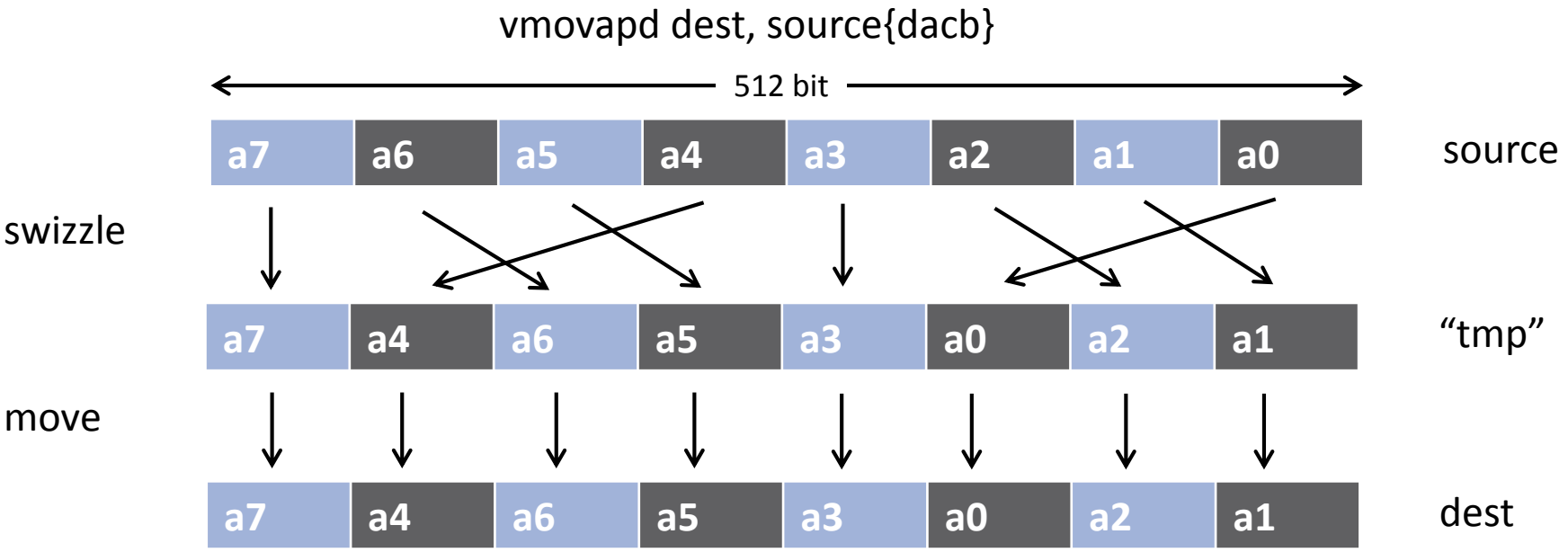
More Powerful SIMD Units

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



More Powerful SIMD Units

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



Auto-vectorization

- Auto vectorization only helps in some cases
  - → Increased complexity of instructions make hard for the compiler to select proper 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
```

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-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++)

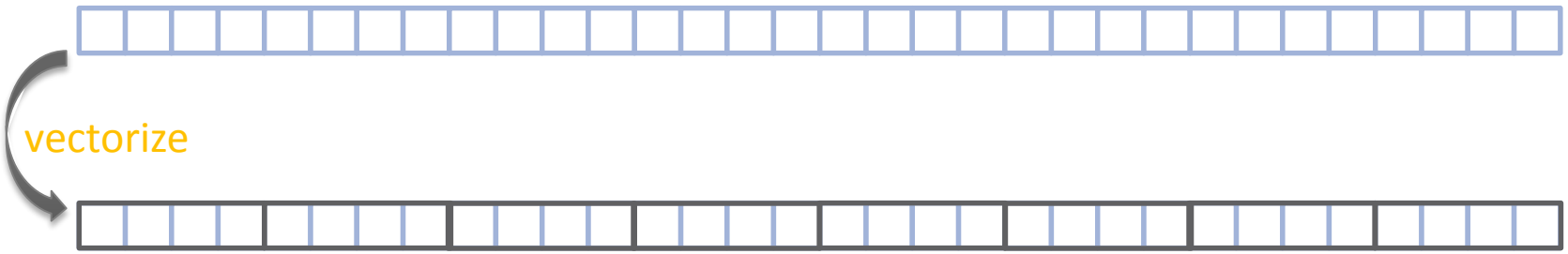
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

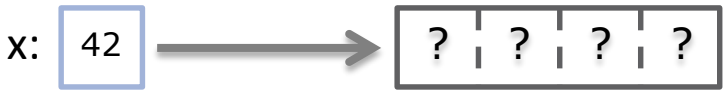
Example

```
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 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 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 simd for
    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 the target device

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definitions-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 simd for
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

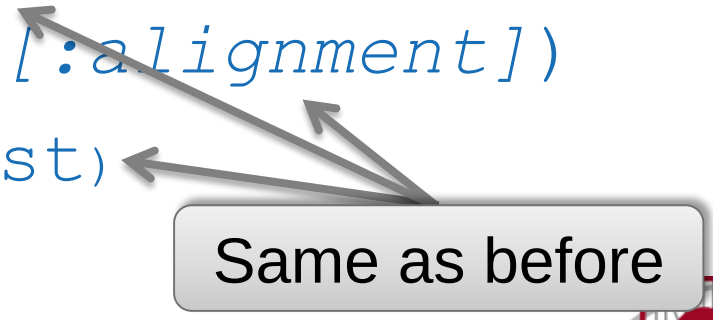
```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
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])`
- `reduction (operator:list)`



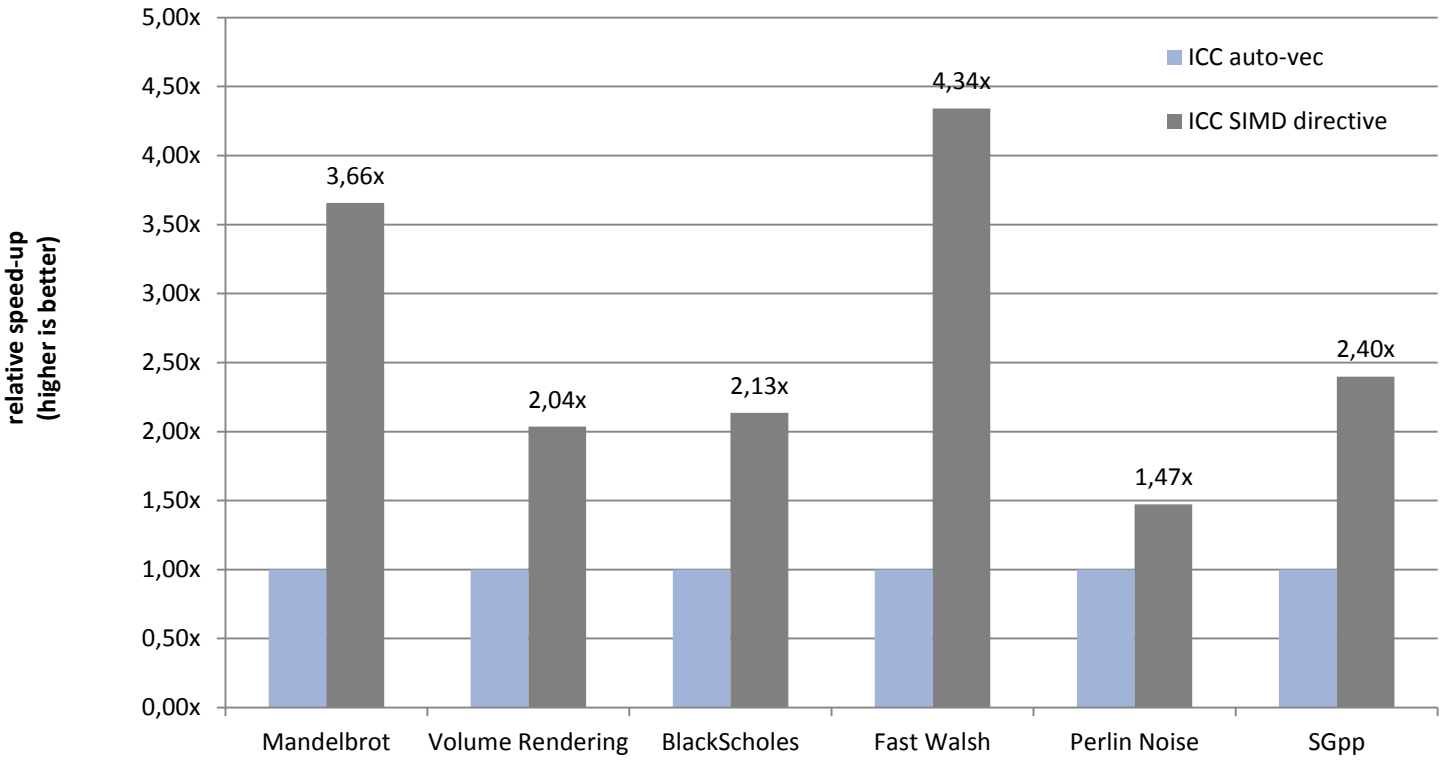
inbranch & notinbranch

```
#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.

Advanced OpenMP Tutorial

Advanced Language Features

Christian Terboven
Michael Klemm
Bronis R. de Supinski

RWTH AACHEN UNIVERSITY
intel
Lawrence Livermore National Laboratory

Members of the OpenMP Language Committee

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

Agenda

- The OpenMP Tasking Model
- Tasking in Detail
- Cancellation
- Miscellaneous OpenMP Features

The OpenMP Tasking Model

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

ISC

The OpenMP Task Construct

C/C++

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

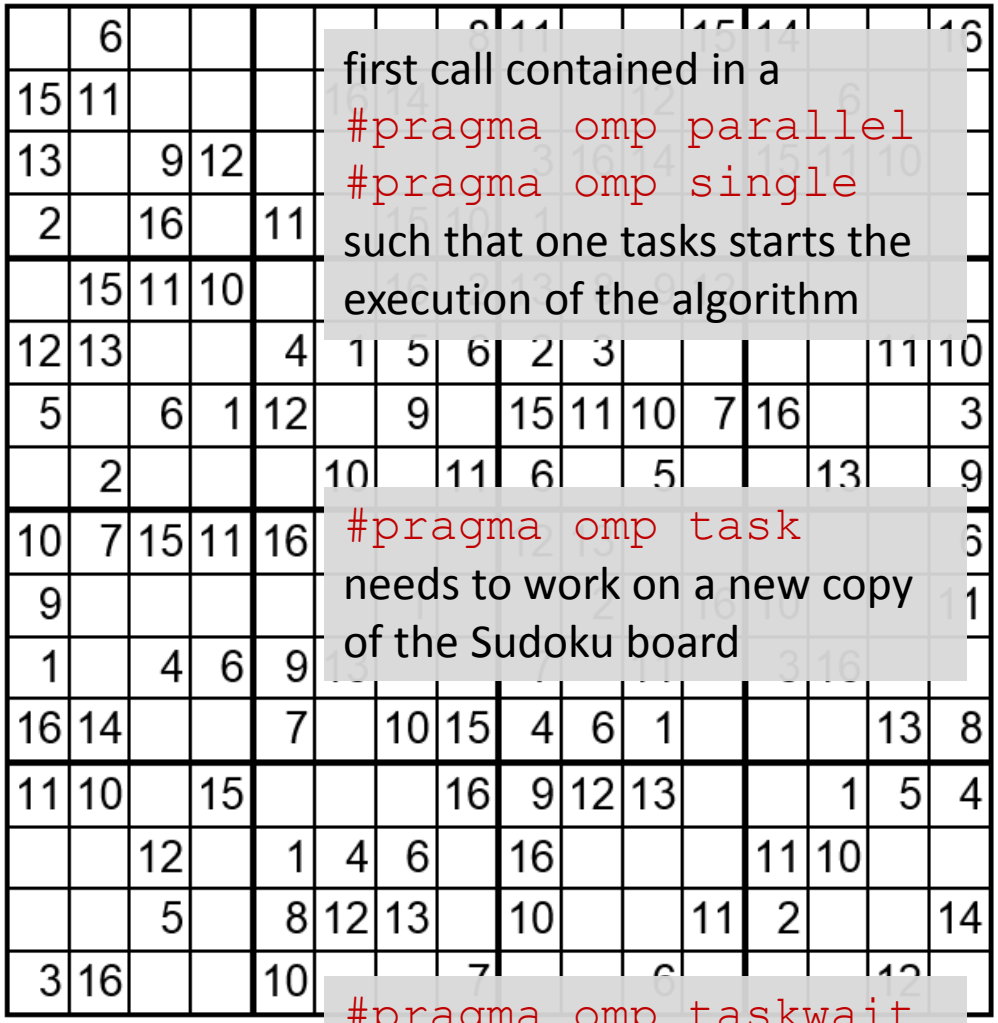
Fortran

```
!$omp task [clause]
... structured block ...
!$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*)

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)

- 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 do not)

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

Parallel Brute-force Sudoku (3/3)

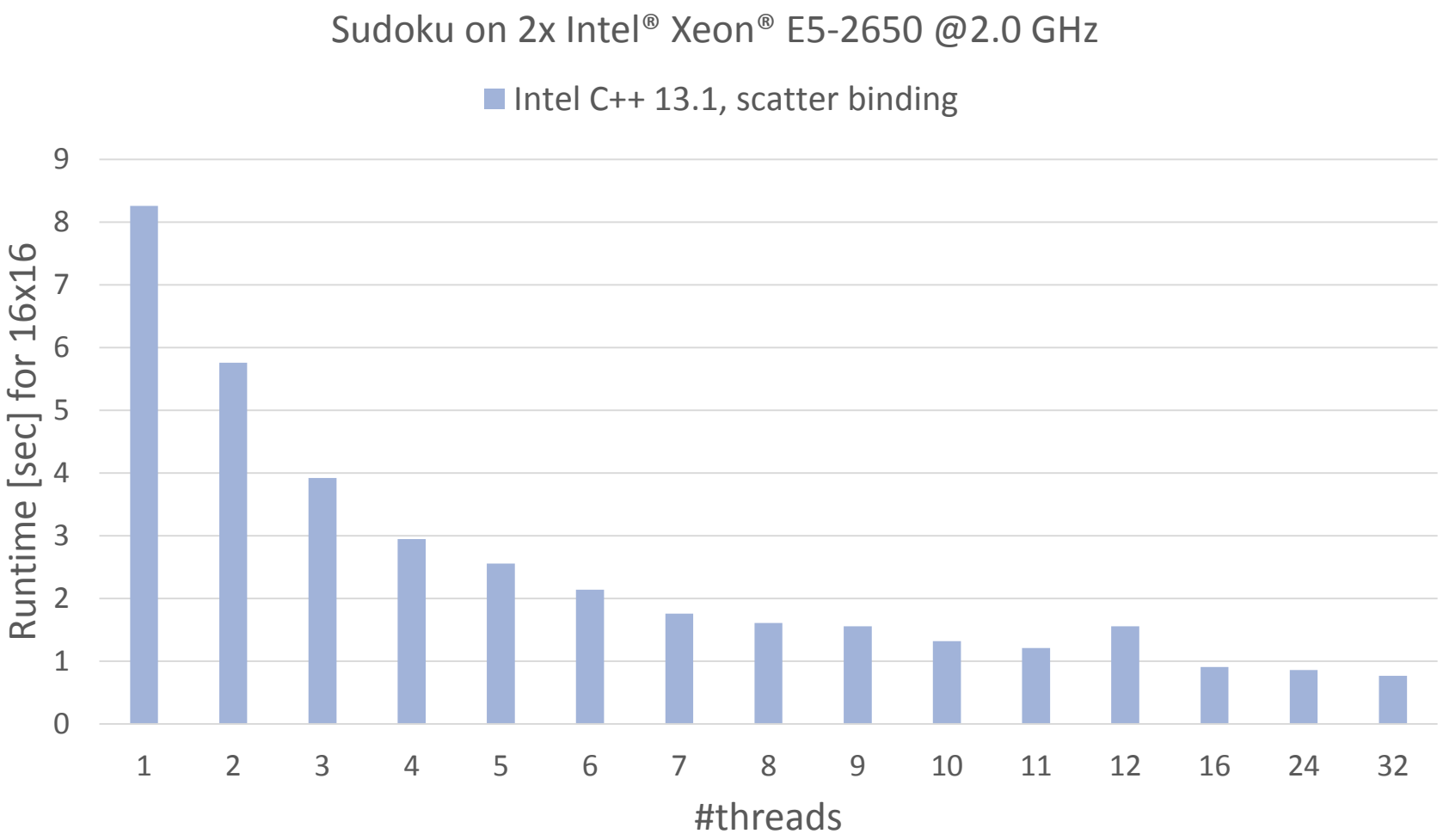
- The actual implementation

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

`#pragma omp taskwait`
wait for all child tasks

Performance Evaluation



Task Sychronization

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 suspends until child tasks are complete
    - → Only direct childs, not descendants!

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

Tasking in Detail

General OpenMP Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - → *private*-list and *shared*-list on Parallel Region
  - → *private*-list and *shared*-list on Worksharing constructs
  - → General default is *shared*, *firstprivate* for Tasks.
  - → Loop control variables on *for*-constructs are *private*
  - → Non-static variables local to Parallel Regions are *private*
  - → *private*: A new uninitialized instance is created for each thread
    - → *firstprivate*: Initialization with Master‘s value / value captured at task creation
    - → *lastprivate*: Value of last loop iteration is written back to Master
  - → Static variables are *shared*

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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

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

Data Scoping Example (2/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

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

Data Scoping Example (3/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

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

Data Scoping Example (4/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // 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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // 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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // 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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

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

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

Task 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 a suspend point
    - → Task creation, task 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

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

Unsafe use of `untied` Tasks

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

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

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

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

The depend Clause

C/C++

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

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

Concurrent Execution w/ Dep.

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

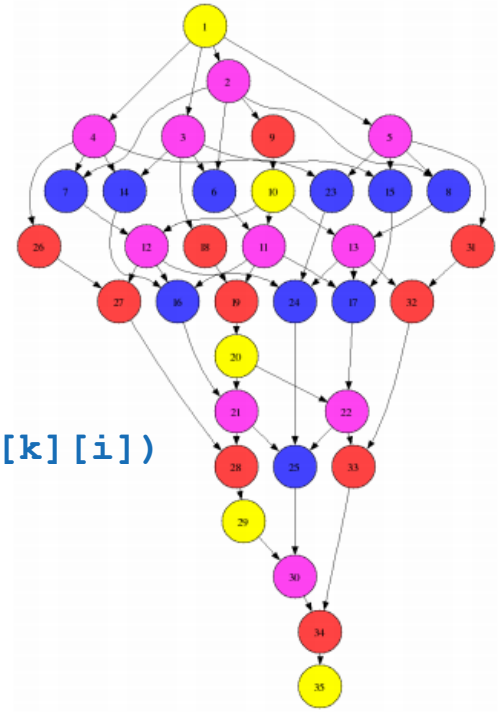
T1 has to be completed before T2 and T3 can be executed.

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
}
```

„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

if, final and mergeable

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

`final` Clause

- For recursive problems that perform task decomposition, stop task creation at a certain depth exposes enough parallelism while reducing the 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++

```
#pragma omp task mergeable
```

Fortran

```
!$omp task mergeable
```

- Personal Note: As of today (09/2012), no compiler or runtime implement final and/or mergeable so that real world application may profit from using them ☹.

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);
        }
}
```

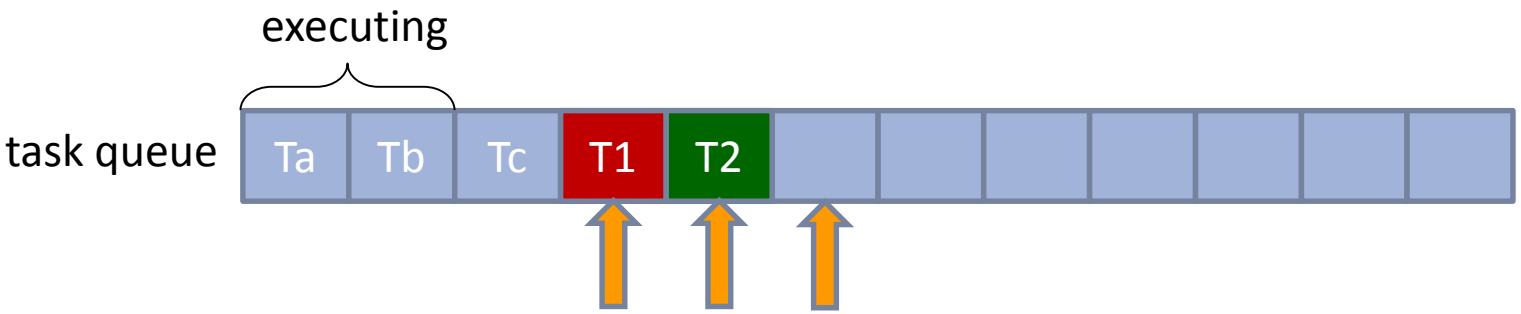
The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.

Implementation Concepts

OpenMP Task Queues

- The runtime system keeps track of a task queue

```
#pragma omp task        // T1
{
    do_something();
}
#pragma omp task        // T2
{
    do_something();
}
```

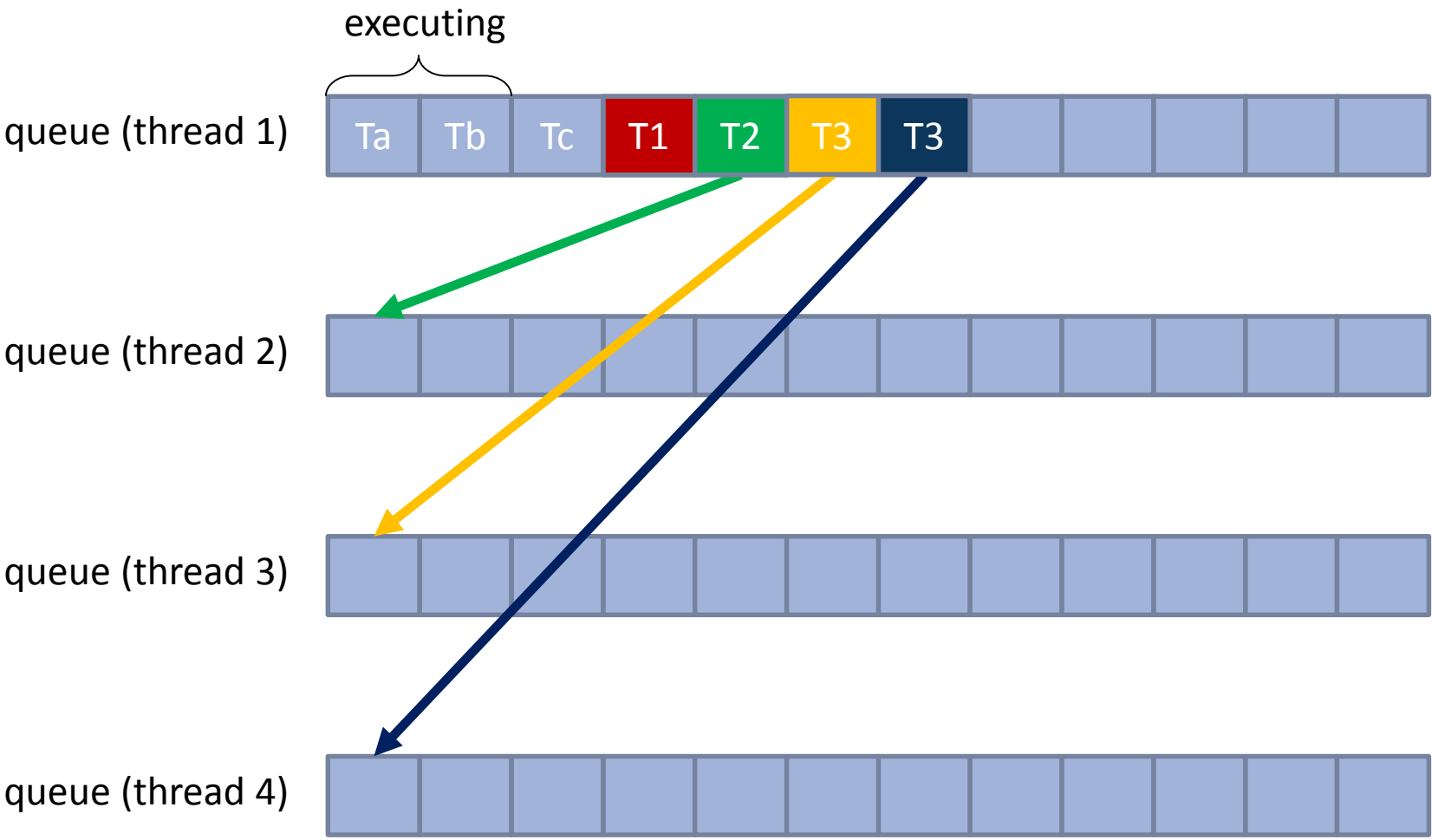


OpenMP Task Queues & NUMA

- **Single task queue has serious limitations**
  - → Single point of contention
  - → NUMA effects can more easily occur (no locality since tasks may be scheduled to run anywhere)

- **Intel Composer XE uses a distributed task queue**
  - → Each thread maintains its own task queue
  - → Newly created tasks are added to the queue of the creator thread
  - → Other threads steal tasks from other threads queues if needed

OpenMP Task Queues & NUMA



Cancellation

OpenMP 3.1 Parallel Abort

- Parallel execution cannot be aborted in OpenMP 3.1
  - → Code regions must always run to completion
  - → (or not start at all)

- Cancellation in OpenMP 4.0 provides a best-effort approach to terminate OpenMP regions
  - → Best-effort: not guaranteed to trigger termination immediately
  - → Triggered “as soon as” possible

Cancellation Constructs

- Two constructs:
  - → Activate cancellation:

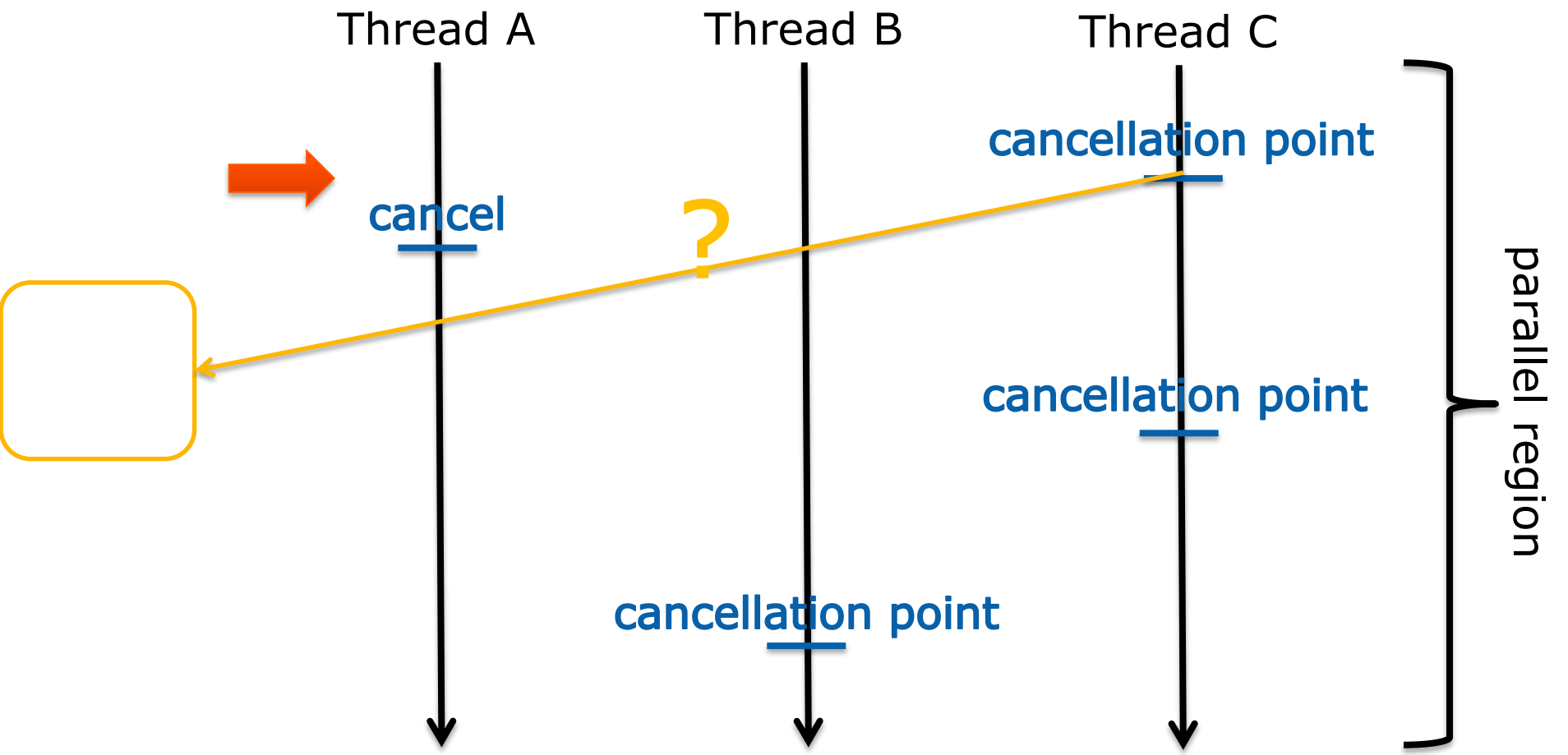
    C/C++: `#pragma omp cancel`

    Fortran: `!$omp cancel`
  - → Check for cancellation:

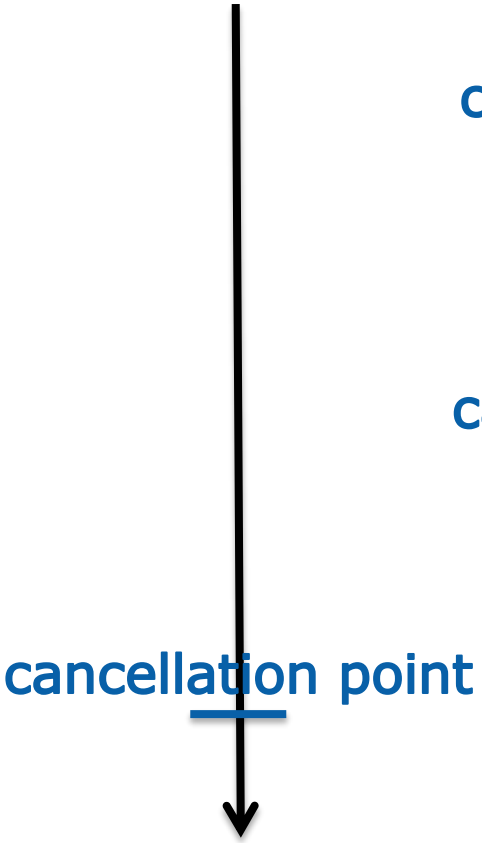
    C/C++: `#pragma omp cancellation point`

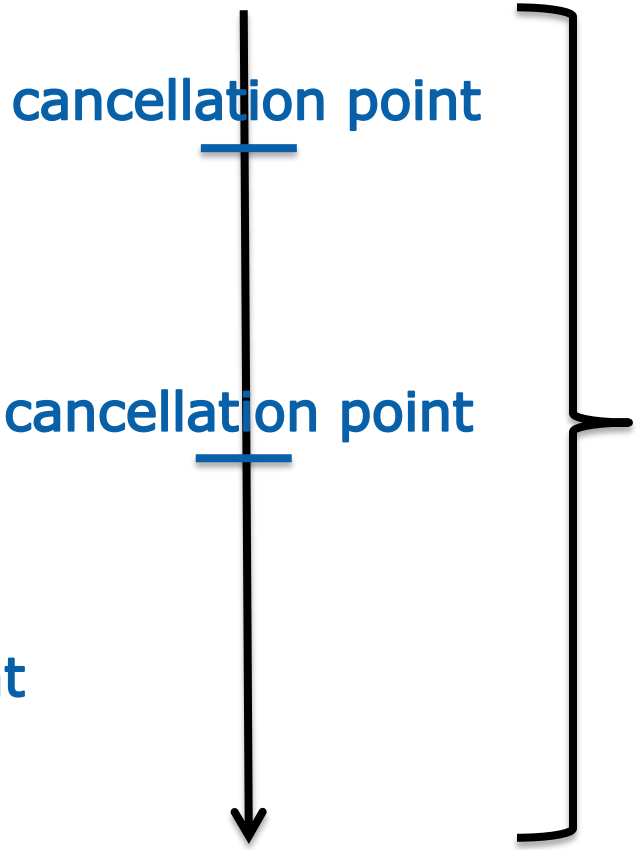
    Fortran: `!$omp cancellation point`
- Check for cancellation only a certain points
  - → Avoid unnecessary overheads
  - → Programmers need to reason about cancellation
  - → Cleanup code needs to be added manually

Cancellation Semantics

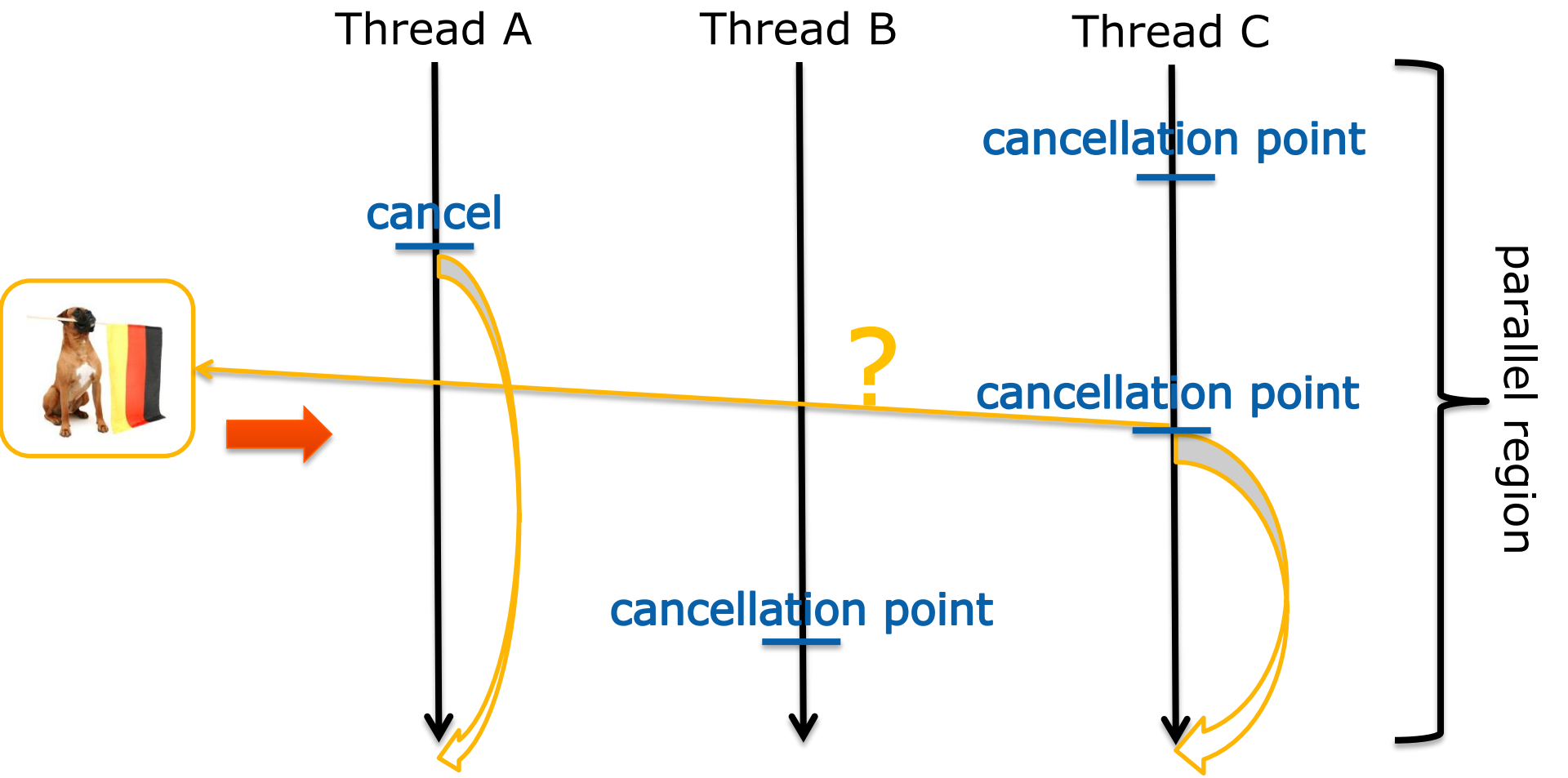


Cancellation Semantics

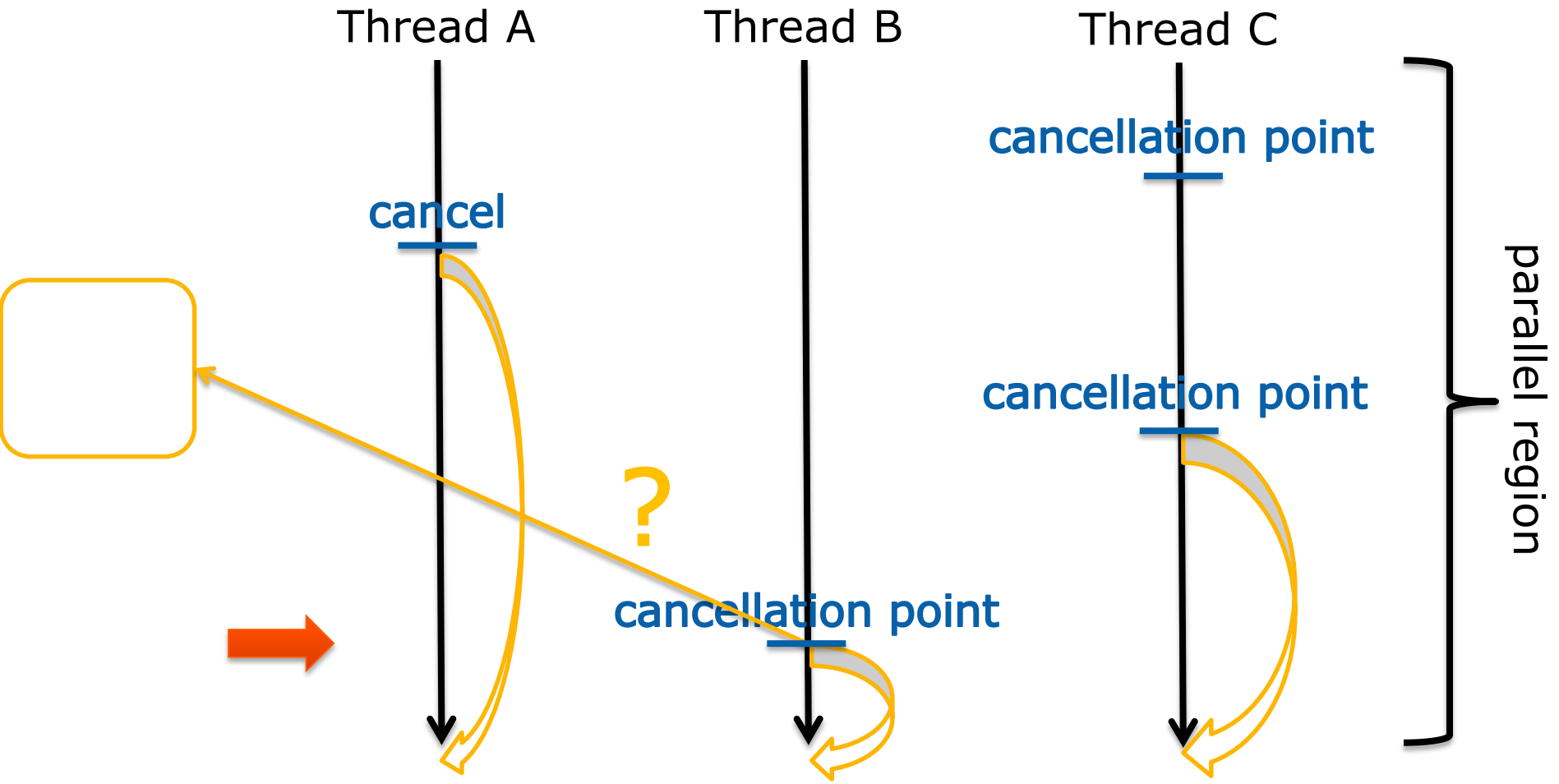




Cancellation Semantics



Cancellation Semantics



cancel Construct

- Syntax:

```
#pragma omp cancel construct-type-clause [ [, ]if-clause]
!$omp cancel construct-type-clause [ [, ]if-clause]
```

- Clauses:

```
parallel
sections
for  (C/C++)
do   (Fortran)
taskgroup
if (scalar-expression)
```

- Semantics
  - → Requests cancellation of the inner-most OpenMP region of the type specified in *construct-type-clause*
  - → Lets the encountering thread/task proceed to the end of the canceled region

cancellation point Construct

- Syntax:

```
#pragma omp cancellation point construct-type-clause
!$omp cancellation point construct-type-clause
```

- Clauses:

`parallel`
`sections`
`for` (C/C++)
`do` (Fortran)
`taskgroup`

- Semantics
  - → Introduces a user-defined cancellation point
  - → Pre-defined cancellation points:
    - → implicit/explicit barriers regions
    - → cancel regions

Cancellation Example

```
subroutine example(n, dim)
  integer, intent(in) :: n, dim(n)
  integer :: i, s, err
  real, allocatable :: B(:)

  err = 0

!$omp parallel shared(err)
  ...
!$omp do private(s, B)
  do i=1, n
    allocate(B(dim(i)), stat=s)

!$omp cancellation point

    if (s .gt. 0) then
!$omp atomic write
      err = s

!$omp cancel do
    endif

    ...
```

```
! ... example continued

! deallocate private array B in
! normal condition
    deallocate(B)
  enddo

! deallcoate any private array that
! has already been allocated
  if (err .gt. 0) then
    if (allocated(B)) then
      deallocate(B)
    endif
  endif

!$omp end parallel
end subroutine
```

Cancellation of OpenMP Tasks

- Cancellation only acts on tasks of group by `taskgroup`
  - → The encountering tasks jumps to the end of its task region
  - → Any executing task will run to completion (or until they reach a `cancellation point` region)
  - → Any task that has not yet begun execution may be discarded (and is considered completed)

- Tasks cancellation also occurs, if a parallel region is canceled.
  - → But not if cancellation affects a worksharing construct.

Task Cancellation Example

```
binary_tree_t* search_tree_parallel(binary_tree_t* tree, int value) {
  binary_tree_t* found = NULL;
#pragma omp parallel shared(found,tree,value)
  {
#pragma omp master
    {
#pragma omp taskgroup
      {
        found = search_tree(tree, value);
      }
    }
  }
  return found;
}
```

Task Cancellation Example

```
binary_tree_t* search_tree(
     binary_tree_t* tree, int value,
     int level) {
  binary_tree_t* found = NULL;
  if (tree) {
    if (tree->value == value) {
      found = tree;
    }
    else {
#pragma omp task shared(found)
      {
        binary_tree_t* found_left;
        found_left =
          search_tree(tree->left, value);
        if (found_left) {
#pragma omp atomic write
          found = found_left;
#pragma omp cancel taskgroup
        }
      }
#pragma omp task shared(found)
      {
        binary_tree_t* found_right;
        found_right =
          search_tree(tree->right, value);
        if (found_right) {
#pragma omp atomic write
          found = found_right;
#pragma omp cancel taskgroup
        }
      }
#pragma omp taskwait
    }
  }
  return found;
}
```

Miscellaneous OpenMP Features

User defined reductions and advanced atomics

3.1 atomic operation additions address an obvious deficiency

- Previously could not capture a value atomically

```
int schedule (int upper) {
   static int iter = 0; int ret;
   ret = iter;
   #pragma omp atomic
      iter++;
   if (ret <= upper) { return ret; }
   else { return -1; }  //no more iters
}
```

- Atomic capture provides the needed functionality

```
int schedule (int upper) {
   static int iter = 0; int ret;
   #pragma omp atomic capture
      ret = iter++;      // atomic capture
   if (ret <= upper) { return ret; }
   else { return -1; }  // no more iters
}
```

- Atomic swap in 4.0 performs capture followed by write
- Added seq_cst clause for atomics in 4.0; removes need for flush…

User Defined Reductions (UDRs) are a major addition

- Use `declare reduction` directive to define operators
- New operators used in reduction clause like predefined ops

```
#pragma omp declare reduction (reduction-identifier :
typename-list : combiner) [identity(identity-expr)]
```

- `reduction-identifier` gives a name to the operator
  - → Can be overloaded for different types
  - → Can be redefined in inner scopes
- `typename-list` is a list of types to which it applies
- `combiner` expression specifies how to combine values
- `identity` can specify the identity value of the operator
  - → Can be an expression or a brace initializer

A simple UDR example

- Declare the reduction operator

```
#pragma omp declare reduction (merge : std::vector<int> :
   omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

- Use the reduction operator in a `reduction` clause

```
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
  #pragma omp parallel for reduction (merge : filtered)
  for (std:vector<int>::iterator it = v.begin(); it < v.end();
it++)
    if ( filter(*it) )  filtered.push_back(*it);
```

- Private copies created for a reduction are initialized to the identity that was specified for the operator and type
  - → Default identity defined if `identity` clause not present
- Compiler uses `combiner` to combine private copies
  - → `omp_out` refers to private copy that holds combined value
  - → `omp_in` refers to the other private copy

Advanced OpenMP Tutorial

OpenMP for Coprocessors/Accelerators

Christian Terboven
Michael Klemm
Bronis R. de Supinski

RWTH AACHEN UNIVERSITY
intel
Lawrence Livermore National Laboratory

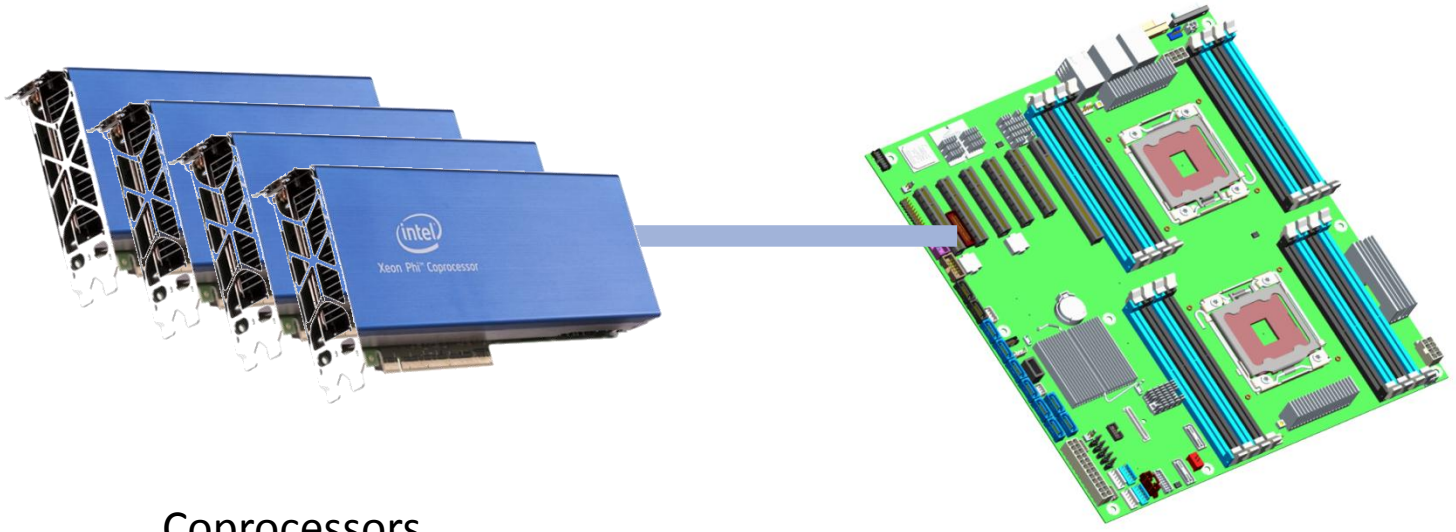
Members of the OpenMP Language Committee

Topics

- Device and execution model
- Control the execution on host and devices
- Examples

Device Model

- OpenMP 4.0 will support accelerators and coprocessors
- Device model:
  - → One host
  - → Multiple accelerators/coprocessors of the same kind



Coprocessors

Host

Terminology

- Device:
  an implementation-defined (logical) execution unit
- League:
  the set of threads teams created by a `teams` construct
- Contention group:
  threads of a team in a league and their descendant threads
- Device data environment:
  Data environment as defined by `target data` or `target` constructs

Terminology

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

`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
```

- Clauses

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

target data Construct

- Create a data device environment
- Syntax (C/C++)

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

- Syntax (Fortran)

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

- Clauses

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

target update Construct

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

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

- Syntax (Fortran)

```
!$omp target data update [clause[[,] clause],…]
structured-block
```

- Clauses

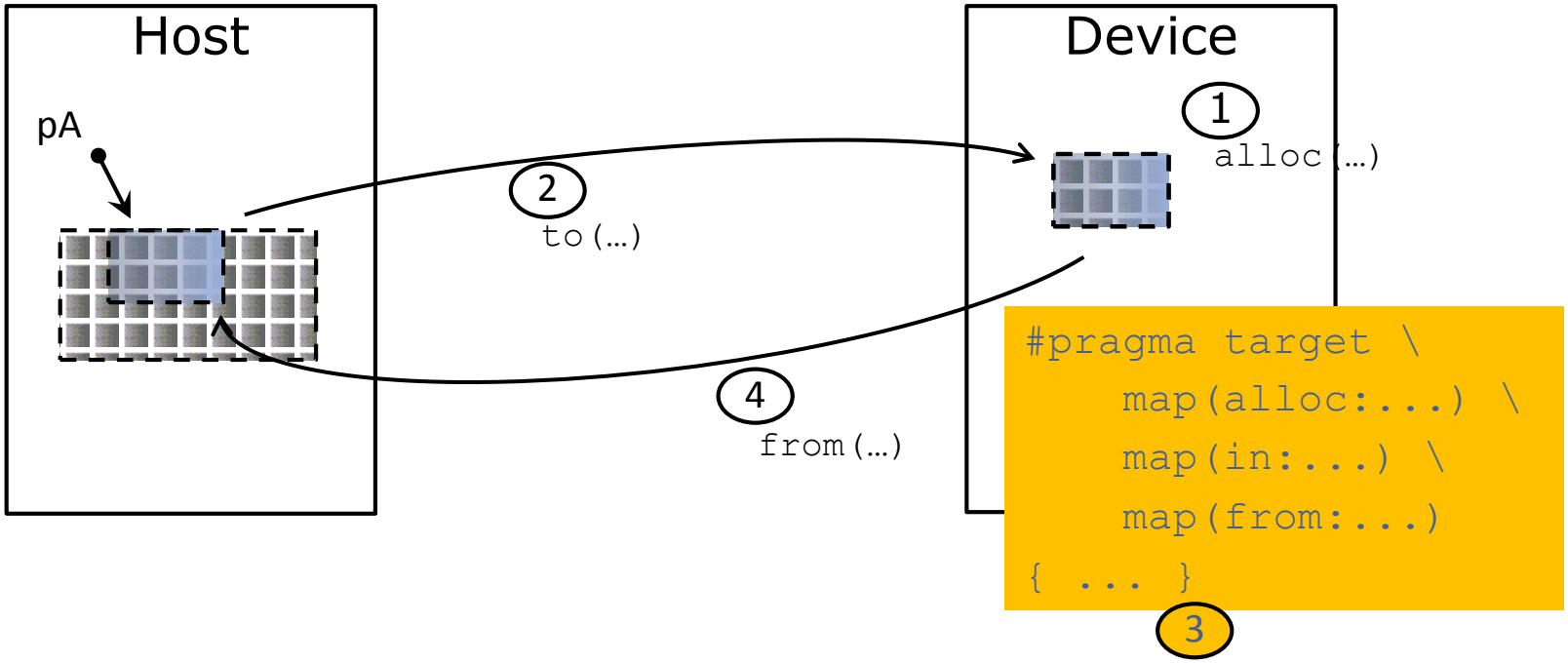
```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

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

Execution Model

- Data environment is lexically scoped
  - → Data environment is destroyed at closing curly brace
  - → Allocated buffers/data are automatically released



Offloading Computation

- Use target construct to
  - → Transfer control from the host to the device
  - → Establish a data environment (if not yet done)
- Host thread waits until offloaded region completed
  - → Use other OpenMP constructs for asynchronicity

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

Data Environments

- Create a data environment to keep data on devices
  - → Avoid frequent transfers or overlap computation/comm.
  - → Pre-allocate temporary fields

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

    do_some_other_stuff_on_host();

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

`target declare` Construct

- Declare one or more functions to also be compiled for the target device

- Syntax (C/C++):

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

- Syntax (Fortran):

```
!$omp declare target [(proc-name-list | list)]
```

Prepare Functions for Device

- The tagged functions will be compiled for
  - → Host execution (as usual)
  - → Target execution (to be invoked from offloaded code)

```
#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
```

```
some_computation:
   ...
   movups %xmm2, (%r15)
   movups %xmm3, (%rbx)
   ...
final_computation:
   ...
```

host functions

```
some_computation_device:
   ...
   vprefetch0 64(%r15)
   vaddps %zmm7, %zmm6, %zmm9
   ...
final_computation_device:
   ...
```

device functions

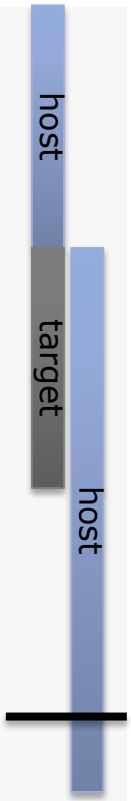
Explicit Data Transfers

```
#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);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

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

Asynchronous Offloading

- Use existing OpenMP features to implement asynchronous offloads.

```
#pragma omp parallel sections
{
#pragma omp task
  {
#pragma omp target map(in:input[:N]) map(out:result[:N])
#pragma omp parallel for
    for (i=0; i<N; i++) {
      result[i] = some_computation(input[i], i);
    }
  }
#pragma omp task
  {
    do_something_important_on_host();
  }
#pragma omp taskwait
}
```



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

`teams` Construct – Restrictions

- Creates a league of thread teams
  - → The master thread of each team executes the `teams` region
  - → Number of teams is specified with `num_teams()`
  - → Each team executes `num_thread()` threads

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`

SAXPY: Serial (host)

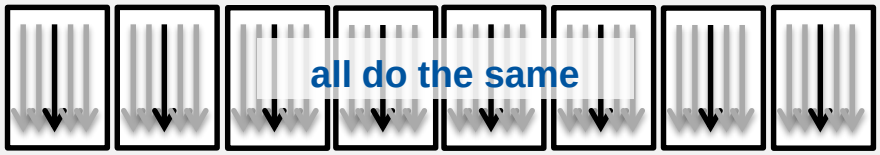
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y









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

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

SAXPY: Serial (host)

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
  {




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

SAXPY: Coprocessor/Accelerator

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
  {
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(nthreads)
```




```
  for (int i = 0; i < n; i += num_blocks){
    for (int j = i; j < i + num_blocks; j++) {
        y[j] = a*x[j] + y[j];
  } }
  }
  free(x); free(y); return 0;
}
```

distribute Construct

- Syntax (C/C++):

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

- Syntax (Fortran):

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

- Clauses

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

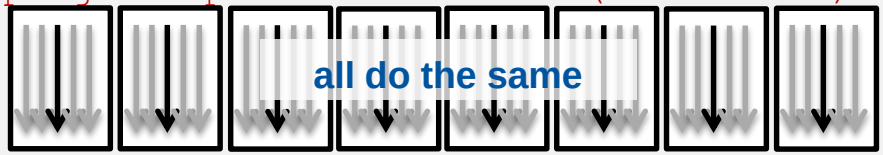
`distribute` Construct

- New kind of worksharing construct
  - → Distribute the iterations of the associated loops across the master threads of a `teams` construct
  - → No implicit barrier at the end of the construct

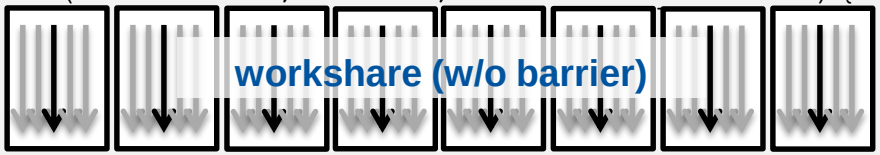
- `dist_schedule(`*`kind[, chunk_size]`*`)`
  - → If specified scheduling kind must be static
  - → Chunks are distributed in round-robin fashion of chunks with size *`chunk_size`*
  - → If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

SAXPY: Coprocessor/Accelerator

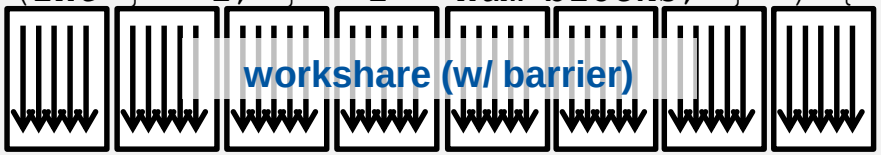
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y
  // Run SAXPY TWICE
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```




```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```




```
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
```




```
        y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

Combined Constructs

- The distribution patterns can be cumbersome

- OpenMP 4.0 defines combined 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

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

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

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

SAXPY: Combined Constructs

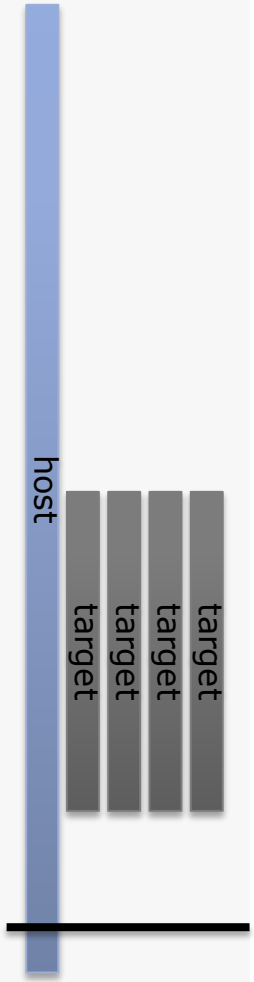
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target map(to:x[0:n]) map(tofrom:y)
  {
#pragma omp teams distribute parallel for \
        num_teams(num_blocks) num_threads(bsize)
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }

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

Additional 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);`
- **Environment variable**
  - **Control default device through** `OMP_DEFAULT_DEVICE`
  - **Accepts a a non-negative integer value**

Multi-device Example

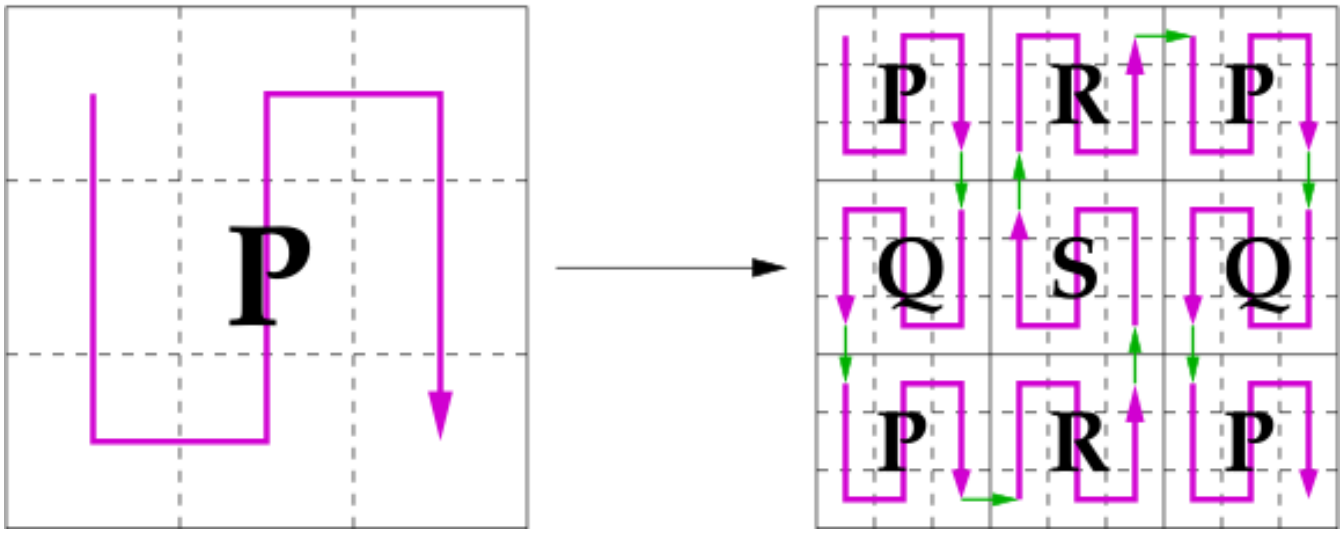
```
int num_dev = omp_get_num_devices();
int chunksz = length / num_dev;
assert((length % num_dev) == 0);
#pragma omp parallel sections firstprivate(chunksz,num_dev)
{
  for (int dev = 0; dev < NUM_DEVICES; dev++) {
#pragma omp task firstprivate(dev)
    {
      int lb = dev * chunksz;
      int ub =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



Final Example: TifaMMy

- Open source project
  - → http://sourceforge.net/projects/tifammy/
- Cache-oblivious matrix operations
  - → Freely adapts to execution platform
- Space filling curves implemented as recursion
  - → Good example for OpenMP tasks
- Vectorized kernels
  - → Using platform-specific intrinsic functions

Space Filling Curve



- Peano curves to cover matrix
- Recursive pattern
  - → Cut block into 3x3 sub-blocks
  - → Repeat recursively until blocks fits the cache/register file

Space Filling Curve: Tasking

```
void MulAdd_MM_PPP_par_omp3(int l_ind_a, int l_ind_b, int l_ind_c,
                            int rec_level) {

  if( rec_level == 1 ) {
    // stop the recursive descent
  }
  else {
    // do the recursion
#pragma omp task
    {
       do_MulAdd_MM_PPP_par_omp3(l_ind_a  , l_ind_b  , l_ind_c, rec_level-1);
       do_MulAdd_MM_RQP_par_omp3(l_ind_a+5, l_ind_b+1, l_ind_c, rec_level-1);
       do_MulAdd_MM_PPP_par_omp3(l_ind_a+6, l_ind_b+2, l_ind_c, rec_level-1);
    }
#pragma omp task
    {
       do_MulAdd_MM_QPQ_par_omp3(l_ind_a+1, l_ind_b  , l_ind_c+1, rec_level-1);
       do_MulAdd_MM_SQQ_par_omp3(l_ind_a+4, l_ind_b+1, l_ind_c+1, rec_level-1);
       do_MulAdd_MM_QPQ_par_omp3(l_ind_a+7, l_ind_b+2, l_ind_c+1, rec_level-1);
    }
    // seven more task constructs
#pragma omp taskwait
  }
}
```

Space Filling Curve: Tasking

- Start the recursion to-level “PPP” block
  - → Top-level “PPP” block is the whole matrix

```
void Multiplication( int nThreads = num_CPUs )
{
#pragma omp parallel num_threads(nThreads)
  {
#pragma omp single nowait
    {
       MulAdd_MM_PPP_par_omp3(0, 0, 0, (int)iteration);
    }
  }
}
```

Offloading to Device

```
#pragma omp target device(0) map(to:A_SIMA[0:sizeSIMA_A]) \
    map(to:B_SIMA[0:sizeSIMA_B]) map(C_SIMA[0:sizeSIMA_C]) \
    map(to:sizeSIMA_A,sizeSIMA_B,sizeSIMA_C)
{
  TifaMMy::TifaMMyMatrixInfo MatrixInfoA;
  TifaMMy::TifaMMyMatrixInfo MatrixInfoB;
  TifaMMy::TifaMMyMatrixInfo MatrixInfoC;

  // fill in matrix information for TifaMMy, e.g.
  // ...
  MatrixInfoA.sizeSIMA = sizeSIMA_A;
  MatrixInfoA.pointerSIMA = (void*)A_SIMA;
  // ...

  // construct TifaMMy matrixes
  TifaMMy::TifaMMyDenseMatrixFloat* A_f =
    new TifaMMy::TifaMMyDenseMatrixFloat ( MatrixInfoA );
  TifaMMy::TifaMMyDenseMatrixFloat* B_f =
    new TifaMMy::TifaMMyDenseMatrixFloat ( MatrixInfoB );
  TifaMMy::TifaMMyDenseMatrixFloat* C_f =
    new TifaMMy::TifaMMyDenseMatrixFloat ( MatrixInfoC );

  // run the matrix-matrix multiplication
  C_f->Multiplication(A_f, B_f);
}
```

OpenMP 4.0 Capabilities

Feature	OpenACC	Intel® LEO	OpenMP 4.0
Support for C and C++, Fortran	✓	✓	✓
Support single code base of hetero-machine	✓	✓	✓
Overlap communication and computation	✓	✓	✓
Interoperate with MPI	✓	✓	✓
Interoperate with OpenMP		✓	✓
Offload to GPU	✓	✓	✓
Offload to Intel Xeon Phi Coprocessor		✓	✓
Ability to support all accelerators			✓
Ability to support all GPUs			✓
Ability to support all co-processors			✓
Support for nested parallelism		✓	✓
User-managed memory consistency	✓	✓	✓
Multiple vendor support	✓		✓
Support for dynamic dispatch		✓	✓
Parallel on/off separate from offload		✓	✓
Intel compiler support		2010*	2013
Broad standards body approval			✓
* public product availability was 2012			

Advanced OpenMP Tutorial

The Future of OpenMP

Christian Terboven
RWTH AACHEN UNIVERSITY

Michael Klemm
intel

Bronis R. de Supinski
Lawrence Livermore National Laboratory

Members of the OpenMP Language Committee

OpenMP 4.0 Status, Directions and Schedule

OpenMP 4.0 ratification vote scheduled for July ARB meeting

- End of a long road? A brief rest stop along the way…
- Addresses several major open issues for OpenMP
- Do not break existing code unnecessarily
- Includes 103 passed tickets
  - →Focus on major tickets initially
  - →Builds on two comment drafts (“RC1” and “RC2”)
  - →Many small tickets after RC2, a few large ones
- Final vote scheduled for July 11
- Already starting work on OpenMP 5.0

Overview of major 4.0 additions

- Device constructs
- SIMD constructs
- Cancellation
- Task dependences and task groups
- Thread affinity control
- User-defined reductions
- Initial support for Fortran 2003
- Support for array sections (including in C and C++)
- Sequentially consistent atomics
- Display of initial OpenMP internal control variables

OpenMP 4.0 will provide unified support for a wide range of devices

- Use `target` directive to offload a region should be offloaded

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

- Creates new data environment from enclosing device data environment
- Clauses support data movement and conditional offloading
  - → `device` supports offload to a device other than default
  - → `map` ensures variables accessible on device
    - → Does not assume copies are made – memory may be shared with host
    - → Does not copy if present in enclosing device data environment
  - → `if` supports running on host if amount of work is small
- Other constructs support device data environment
  - → `target data` places `map` list items in device data environment
  - → `target update` ensures variable is consistent in host and device

Several other device constructs support simple offload of full-featured code

- Use `target declare` directive to create device versions

```
#pragma omp declare target
```

  - → Can be applied to functions and global variables
  - → Required for UDRs that use functions and execute on device
- `teams` directive creates multiple teams in a `target` region

```
#pragma omp teams [clause [[,] clause] …]
```

  - → Work across teams only synchronized at end of `target` region
  - → Useful for GPUs (corresponds to thread blocks)
- Use `distribute` directive to run loop across multiple teams

```
#pragma omp distribute [clause [[,] clause] …]
```

- Several combined constructs (post-RC2) simplify device use

Reminiscent of our roots, OpenMP 4.0 will provide portable SIMD constructs

- Use `simd` directive to indicate a loop should be SIMDized

```
#pragma omp simd [clause [[,] clause] …]
```

- Execute iterations of following loop in SIMD chunks
  - → Region binds to the current task, so loop is not divided across threads
  - → SIMD chunk is set of iterations executed concurrently by a SIMD lanes
- Creates a new data environment
- Clauses control data environment, how loop is partitioned
  - → `safelen(`*`length`*`)` limits the number of iterations in a SIMD chunk
  - → `linear` lists variables with a linear relationship to the iteration space
  - → `aligned` specifies byte alignments of a list of variables
  - → `private`, `lastprivate`, `reduction` and `collapse` usual meanings
  - → Would `firstprivate` be useful?

What happens if a SIMDized loop includes function calls?

- Could rely on compiler to handle
  - → Compiler could in-line function to SIMDize its operations
  - → Compiler could try to generate SIMDize version of function
  - → Inefficient default would call function from each SIMD lane
- Provide `declare simd` directive to generate SIMD function

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

  - → Invocation of generated function processes across SIMD lanes
- Clauses control data environment, how function is used
  - → `simdlen(`*`length`*`)` specifies the number of concurrent arguments
  - → `uniform` lists invariant arguments across concurrent SIMD invocations
  - → `inbranch` and `notinbranch` imply always/never invoked in conditional statement
  - → `linear`, `aligned`, and `reduction` are similar to `simd` clauses

The loop SIMD and parallel loop SIMD combine two types of parallelism

- The loop SIMD construct workshares and SIMDizes loop

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

  - → Cannot be specified separately
  - → Loop is first divided into SIMD chunks
  - → SIMD chunks are divided across implicit tasks
  - → Not guaranteed same schedule even with `static` schedule
- Parallel loop SIMD creates a parallel region with a loop SIMD region

```
#pragma omp paralel for simd [clause [[,] clause] …]
```

  - → Purely a convenience that combines separate directives
  - → Analogous to the combined parallel worksharing constructs
  - → Would a parallel SIMD construct (i.e., no worksharing) be useful?

The `declare simd` construct supports SIMD execution of library routines

- Tells compiler to generate SIMD versions of functions

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

#pragma omp simd notinbranch
float distsq (float x, float y) {
   return (x – y) * (x – y);  }
```

- Compile library and use functions in a SIMD loop

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

→ Creates implicit tasks of parallel region
→ Divides loop into SIMD chunks
→ Schedules SIMD chunks across implicit tasks
→ Loop is fully SIMDized by using SIMD versions of functions

4.0 significantly extends initial high-level affinity support of OpenMP 3.1

- Control of nested thread team sizes (in OpenMP 3.1)

```
export OMP_NUM_THREADS=4,3,2
```

- Request binding of threads to places (in OpenMP 3.1)

```
export OMP_PROC_BIND=TRUE
```

- New extensions specify thread locations
  - → Increased choices for `OMP_PROC_BIND`
    - → Can still specify `true` or `false`
    - → Can now provide a list (possible item values: `master`, `close` or `spread`) to specify how to bind implicit tasks of `parallel` regions
  - → Added `OMP_PLACES` environment variable
    - → Can specify abstract names including `threads`, `cores` and `sockets`
    - → Can specify an explicit ordered list of places
    - → Place numbering is implementation defined

Affinity support now supports targeting thread binding to specific parallel regions

- Added a new clause to the `parallel` construct

```
proc_bind(master | close | spread)
```

  - → Overrides `OMP_PROC_BIND` environment variable
  - → Ignored if `OMP_PROC_BIND` is `false`

- New run time function to query current policy

```
omp_proc_bind_t omp_get_proc_bind(void);
```

- New policies determine relative bindings
  - → Assign threads to same place as `master`
  - → Assign threads `close` in place list to parent thread
  - → Assign threads to maximize `spread` across places

OpenMP 4.0 will include initial support for Fortran 2003

- Added to list of base language versions
- Have a list of unsupported Fortran 2003 features
  - → List initially included 24 items (some big, some small)
  - → List has been reduced to 14 items
  - → List in specification reflects approximate priority
  - → Priorities determined by importance and difficulty
- Strategy: Gradually reduce list until full support available in 5.0
  - → Removed procedure pointers, renaming operators on the `USE` statement, `ASSOCIATE` construct, `VOLATILE` attribute and structure constructors
  - → Will support Fortran 2003 object-oriented features next
    - → The biggest issue
    - → Considering concurrent reexamination of C++ support

4.0 adds taskgroup construct to simplify task synchronization

- Adds one easily shown construct

```
#pragma omp taskgroup
{
        create_a_group_of_tasks (could_create_nested_tasks);
}
```

  - → Implicit task scheduling point at end of region; current task is suspended until all child tasks generated in the region and their descendants complete execution
  - → Similar in effect to a deep `taskwait`
    - → 3.1 would require more synchronization, more directives
- More significant tasking extension added concept of task dependence: the `depend` clause

Open Discussion of Possible OpenMP Extensions

We are considering several other topics for OpenMP 5.0 and beyond

- Support for memory affinity
- Refinements to accelerator support
- Transactional memory and thread level speculation
- Additional task/thread synchronization mechanisms
- Completing extension of OpenMP to Fortran 2003
- Interoperability, composability and modularity
- Incorporating tool support