Many-core Programming with OpenMP* 4.x

Dr.-Ing. Michael Klemm
Senior Application Engineer
Software and Services Group
(michael.klemm@intel.com)

*Other brands and names are the property of their respective owners.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015 Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

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

Outline

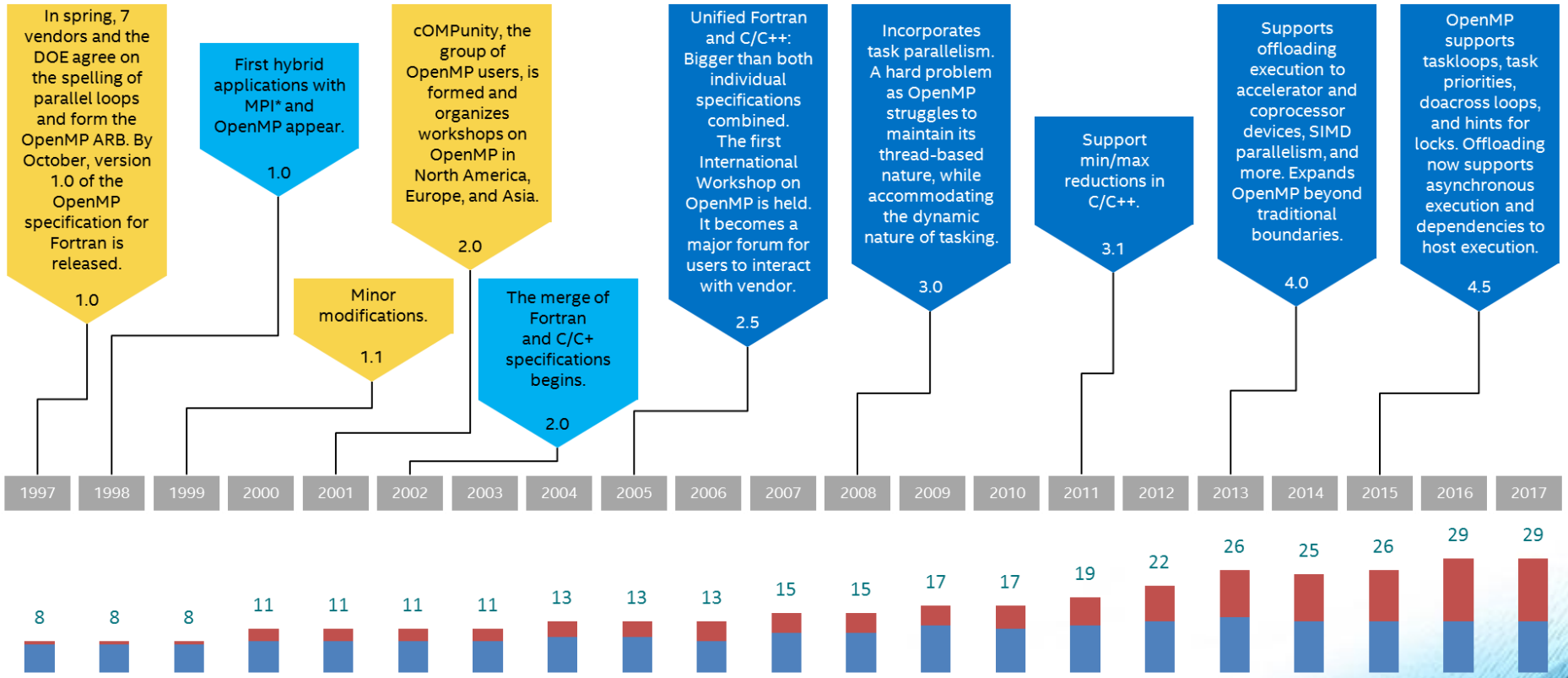
- Very brief Introduction to OpenMP
- OpenMP for Many-core Processors
- OpenMP SIMD Constructs
- Task-generating loops
- Affinity Control

Brief Introduction to OpenMP

OpenMP API

- De-facto standard, OpenMP 4.0 out since July 2013
- API for C/C++ and Fortran for shared-memory parallel programming
- Based on directives (pragmas in C/C++)
- Portable across vendors and platforms
- Supports various types of parallelism

OpenMP History



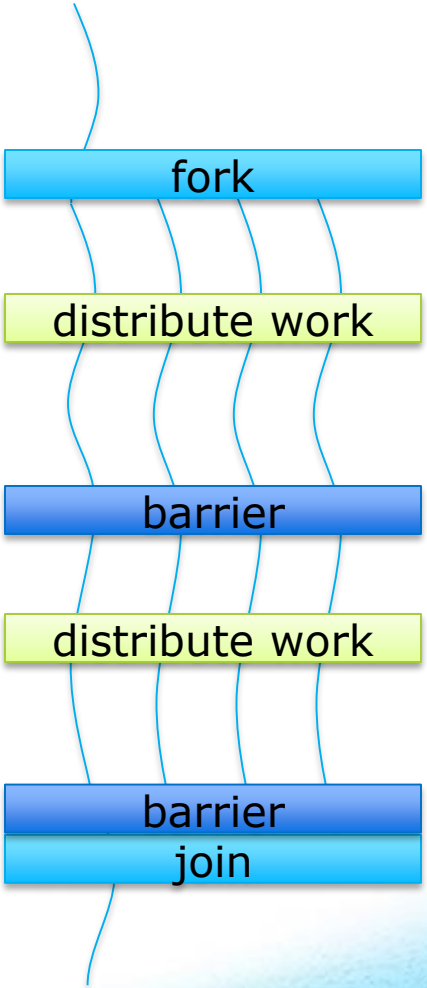
SOFTWARE AND SERVICES ■ Permanent ARB ■ Auxiliary ARB

OpenMP Platform Features

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

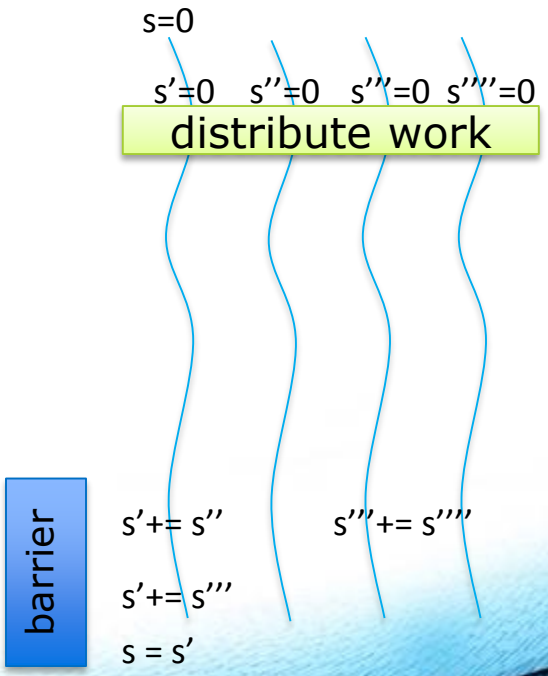
OpenMP 3.0 in Three Slides

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i<N; i++)
    {…}

    #pragma omp for
    for (i = 0; i< N; i++)
    {…}
}
```



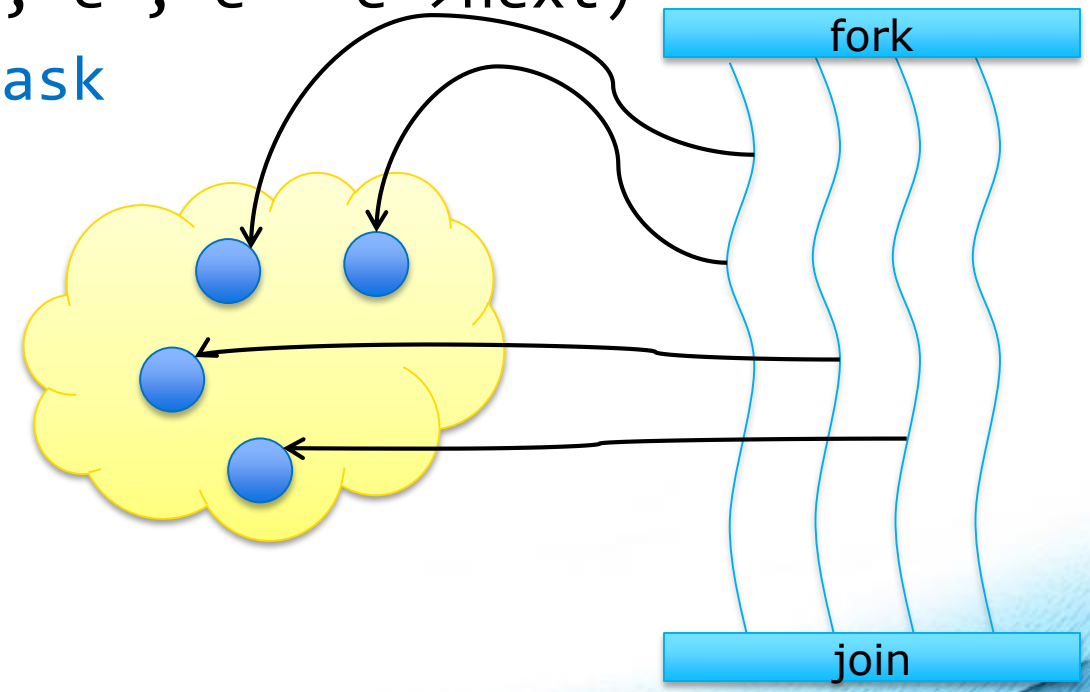
OpenMP 3.0 in Three Slides /2

```
double a[N];
double l,s = 0;
#pragma omp parallel for reduction(+:s) private(l) \
                         schedule(static,4)

for (i = 0; i<N; i++)
{
   l = log(a[i]);
   s += l;
}
```



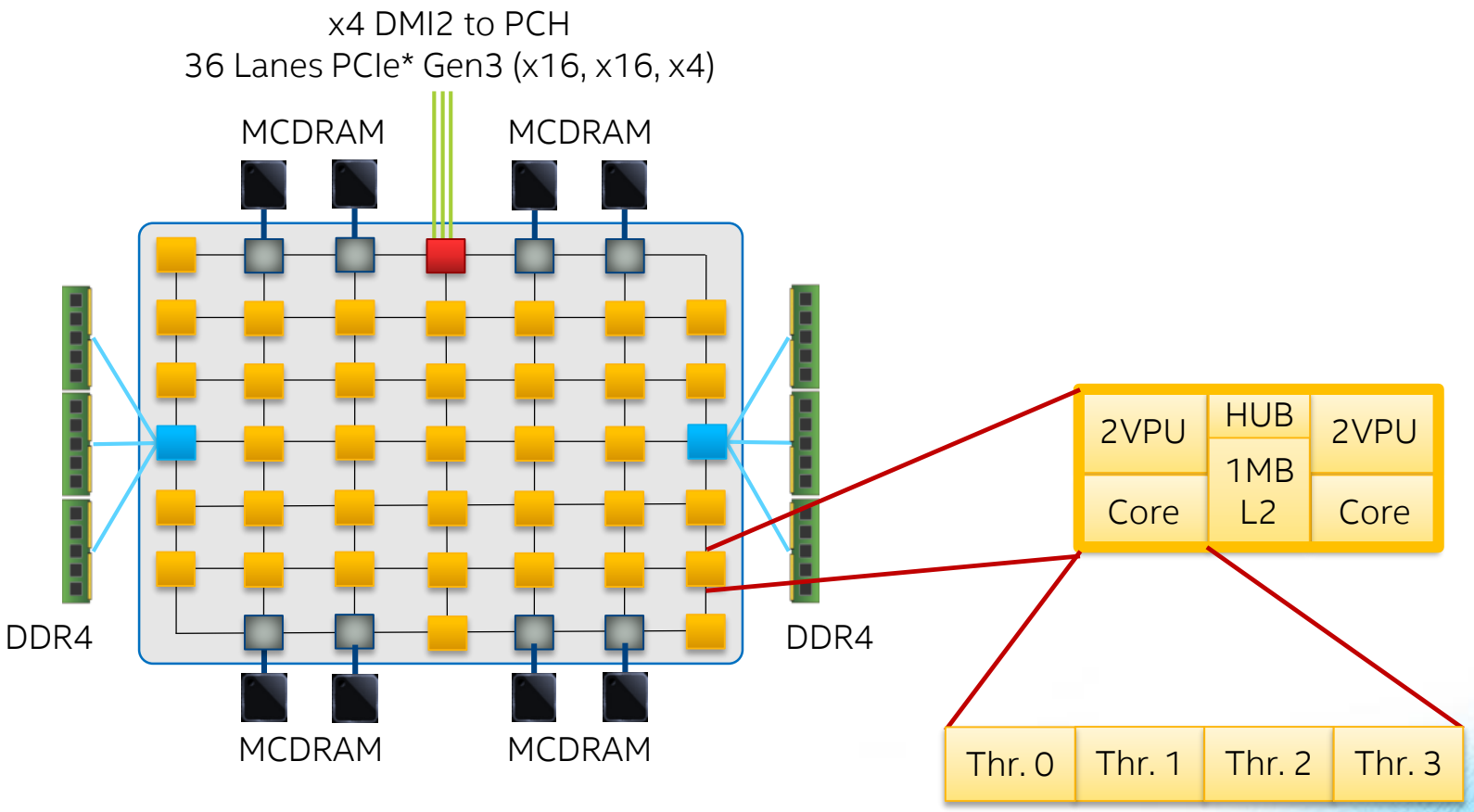
OpenMP 3.0 in Three Slides /3

```
#pragma omp parallel
#pragma omp single
for(e = l->first; e ; e = e->next)
    #pragma omp task
      process(e);
```

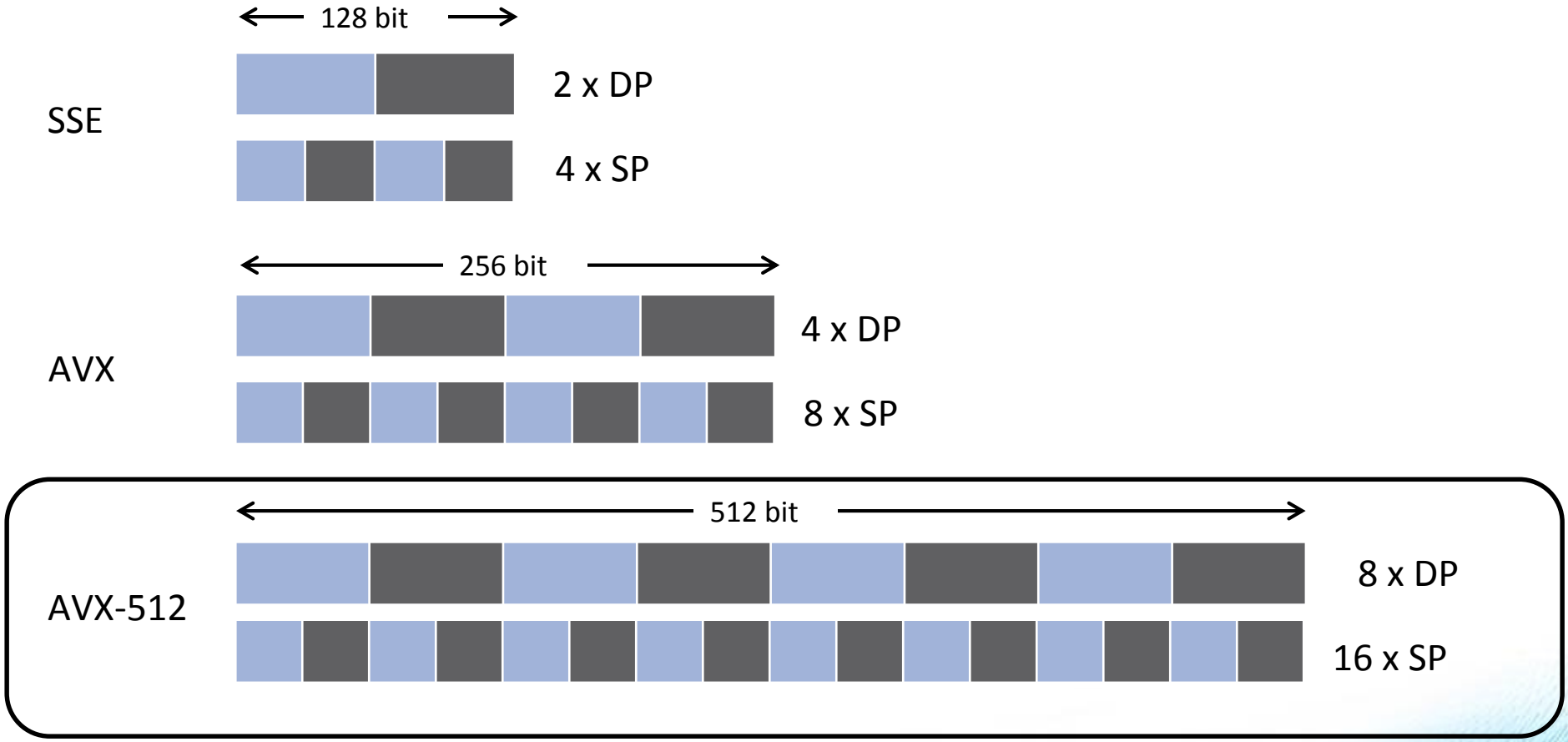


OpenMP for Many-core Processors

Intel Xeon Phi Tile Architecture



SIMD on Intel® Architecture



Good News and Bad News!

- Goods news are:
  - OpenMP has already everything you need to get to good many-core performance
  - You can combine individual OpenMP features to match your code with the underlying hardware structure

- Bad news are:
  - Simplistically adding pragmas to loops does no longer cut it!
  - You need to think about parallelism at much higher levels
  - My above statement was a bit of a lie…
    - Memory allocation still missing from OpenMP
    - Still requires use of non-OpenMP APIs

OpenMP SIMD Constructs

Why Auto-vectorizers Fail

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

Data Dependencies

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

FLOW

```
s1: a = 40

    b = 21

s2: c = a + 2
```

ANTI

```
    b = 40

s1: a = b + 1

s2: b = 21
```

Loop-Carried Dependencies

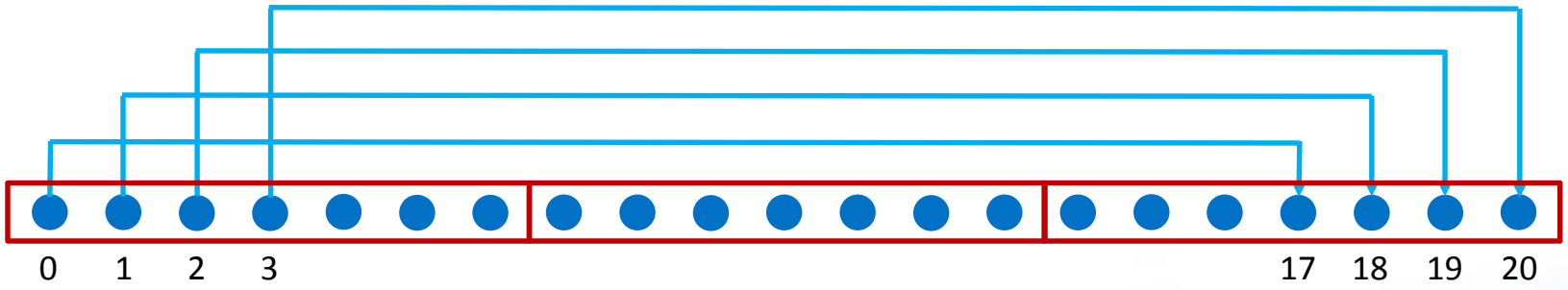
- Dependencies may occur across loop iterations
  - Loop-carried dependency
- The following code contains such a dependency:

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

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

Loop-Carried Dependencies

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



Example: Loop not Countable

- "Loop not Countable" plus "Assumed Dependencies"

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

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

In a Time before OpenMP 4.0

- Programmers had to rely on auto-vectorization…
- … or to use vendor-specific extensions
  - Programming models (e.g., Intel® Cilk™ Plus)
  - Compiler pragmas (e.g., `#pragma vector`)
  - Low-level constructs (e.g., `_mm_add_pd()`)

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

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

OpenMP 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++)

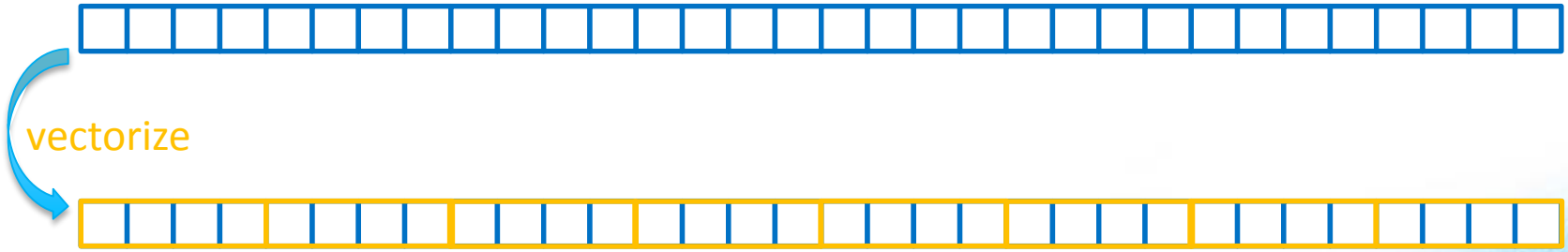
```
#pragma omp [for] simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

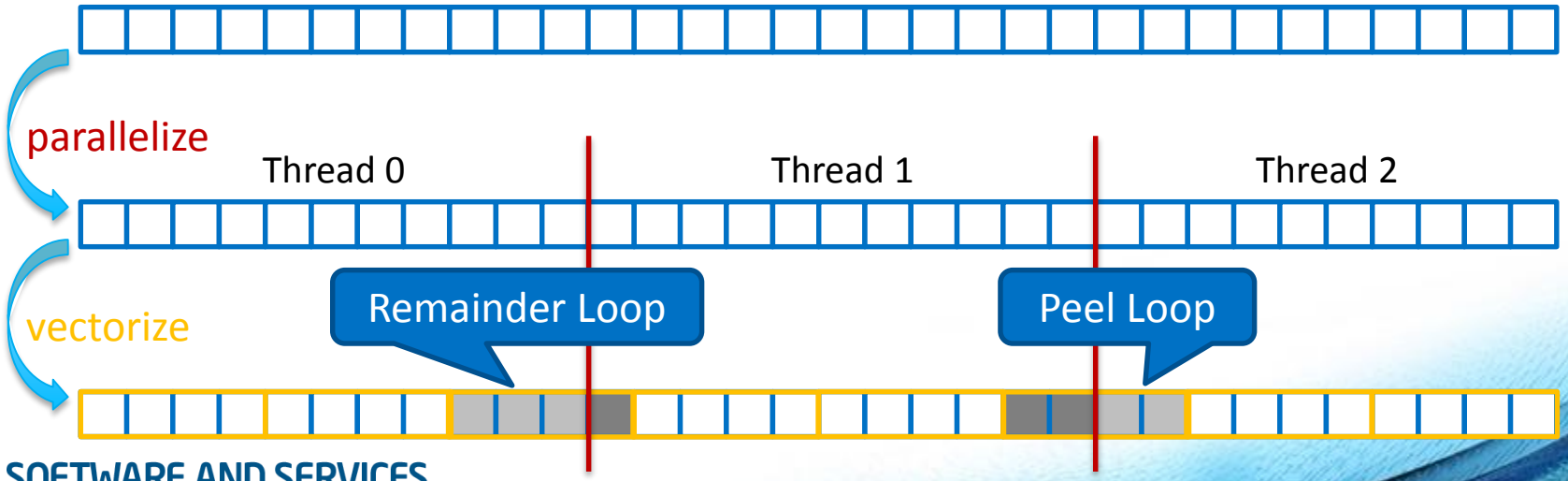
Example

```
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;
}
```



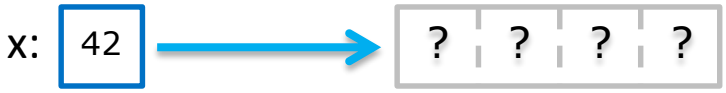
Example: Worksharing w/ SIMD

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



Data Sharing Clauses

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



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

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

SIMD Loop Clauses

- `safelen (length)`
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length
- `linear (list[:linear-step])`
  - The variable's value is in relationship with the iteration number

    $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- `collapse (n)`

Be Careful What You Wish For…

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

- You should choose chunk sizes that are multiples of the SIMD length
  - Remainder loops are not triggered
  - Likely better performance
- In the above example …
  - and AVX-512, the code will only execute the remainder loop!
  - and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

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

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

SIMD Function Vectorization

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

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

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

SIMD Function Vectorization

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

- Syntax (C/C++):

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

- Syntax (Fortran):

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

SIMD Function Vectorization

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

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

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

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

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

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

SIMD Function Vectorization

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

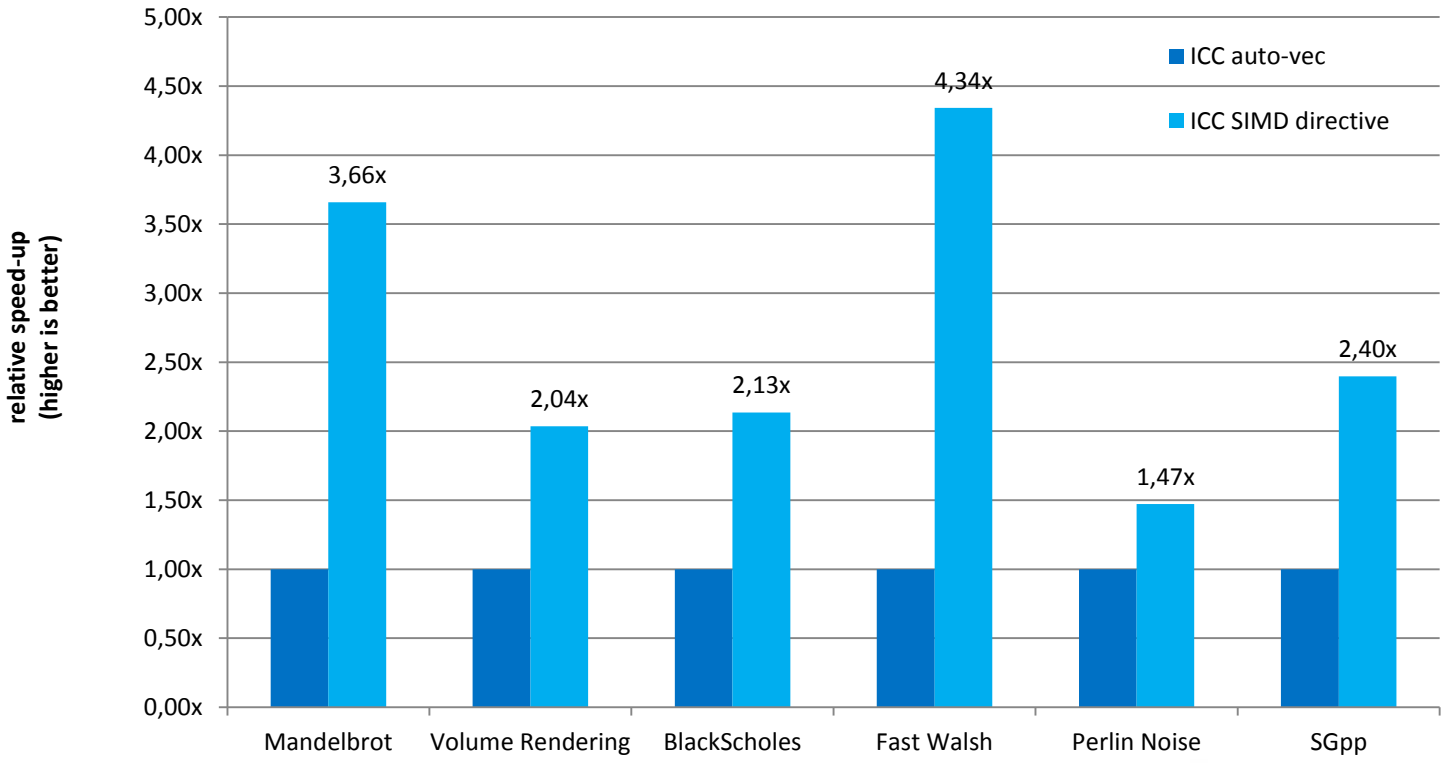
inbranch & notinbranch

```
#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.

Task-generating Loops

Issues with Traditional Worksharing

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

```
void example() {
#pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // dgemm is either parallel or sequential
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);

}   }
```

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

Issues with Traditional Worksharing /2

- Worksharing constructs do not compose well
- Pathological example: load imbalance

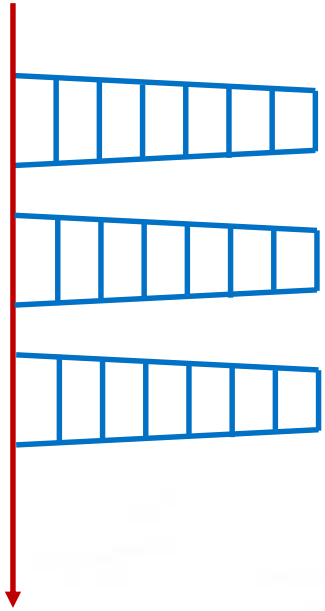
```
void load_imbalance() {
    long_running_task() // can execute concurrently

    for (int i = 0; i < N; i++) { // can execute concurrently
        for (int j = 0; j < M; j++) {
            loop_body(i, j);
        }
    }
}
```

- Writing such code requires
  - nested parallelism,
  - manual, non-portable fine-tuning, and
  - a lot of care to get the load balance right.

Ragged Fork/Join

- Traditional worksharing can lead to ragged fork/join patterns

```
void example() {

    compute_in_parallel(A);

    compute_in_parallel_too(B);

    cblas_dgemm(…, A, B, …);

}
```



Example: Sparse CG

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

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

The `taskloop` Construct

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

- Syntax (C/C++)

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

- Syntax (Fortran)

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

Clauses for `taskloop` Construct

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

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

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

Example: `task` and `taskloop`

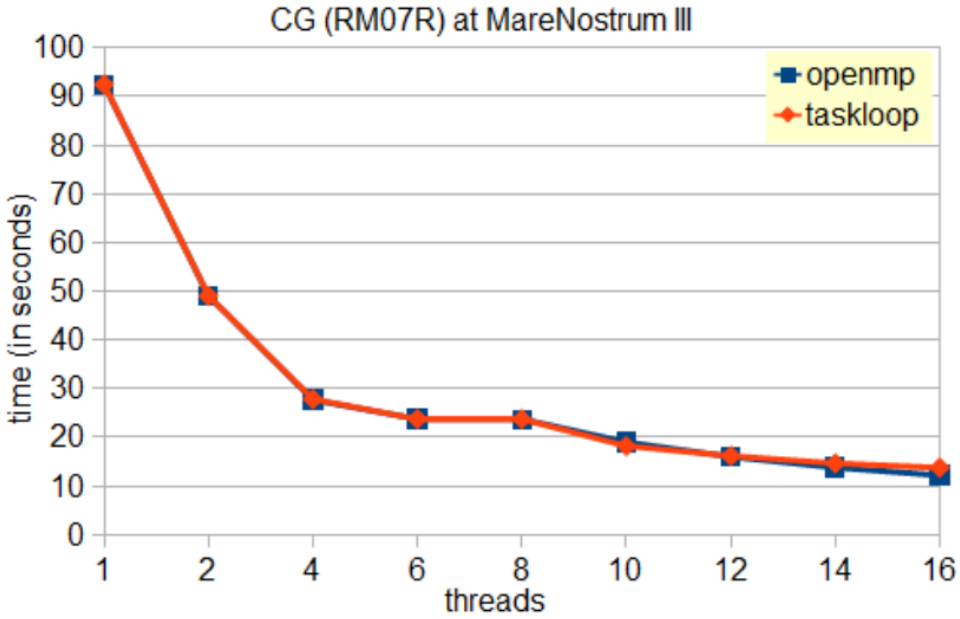
```
void load_imbalance() {
#pragma omp taskgroup
    {
#pragma omp task
        long_running_task() // can execute concurrently

#pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++) { // can execute concurrently
            for (int j = 0; j < M; j++) {
                loop_body(i, j);
            }
        }
    }
}
```

Example: Sparse CG, `taskloop`

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

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

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

Performance of Sparse CG w/ Tasks

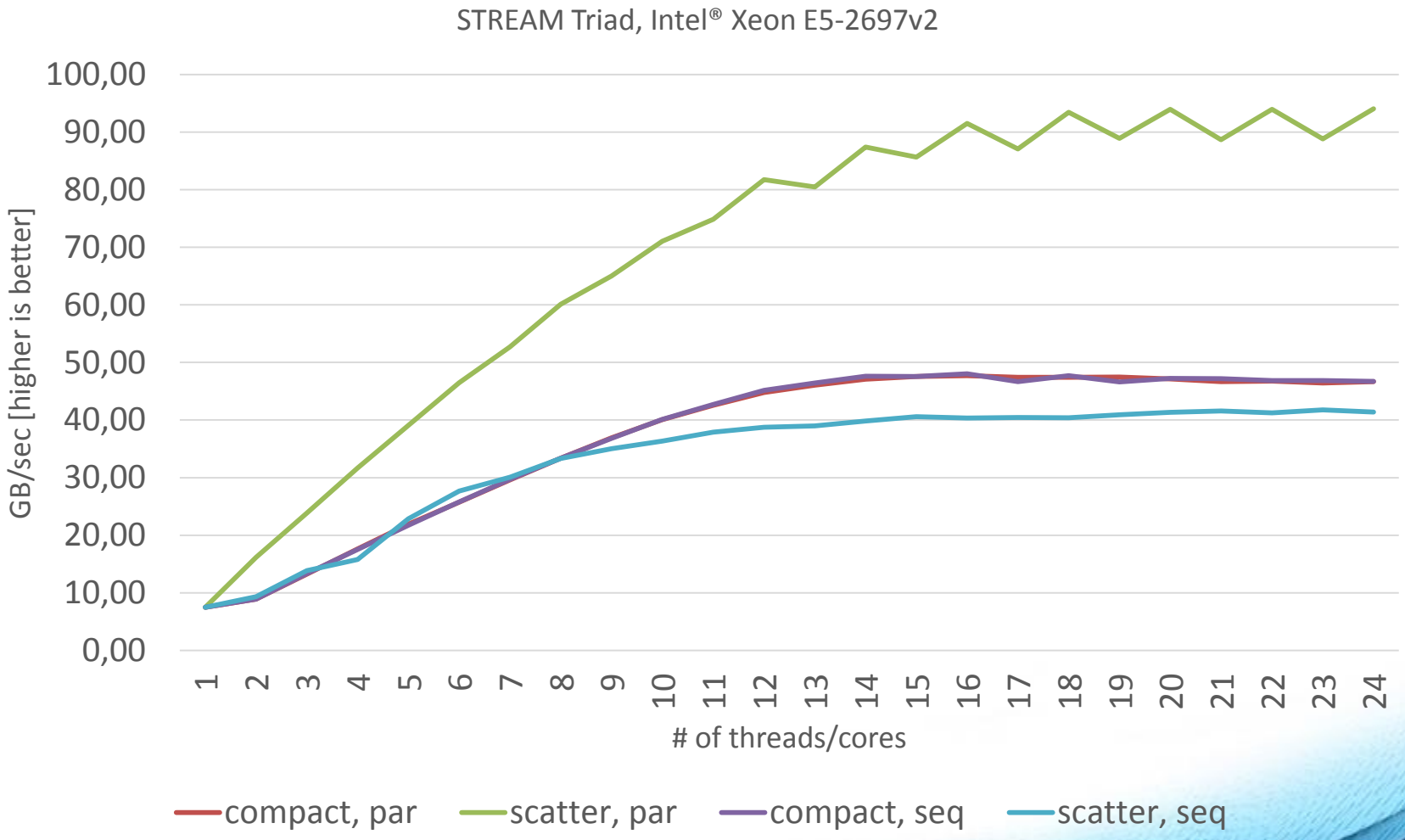


X. Teruel, M. Klemm, K. Li, X. Martorell, S.L. Olivier, and C. Terboven. A Proposal for Task-Generating Loops in OpenMP. In A.P. Rendell et al., editor, International Workshop on OpenMP, pages 1-14, Canberra, Australia, September 2013. LNCS 8122

NUMA is here to Stay…

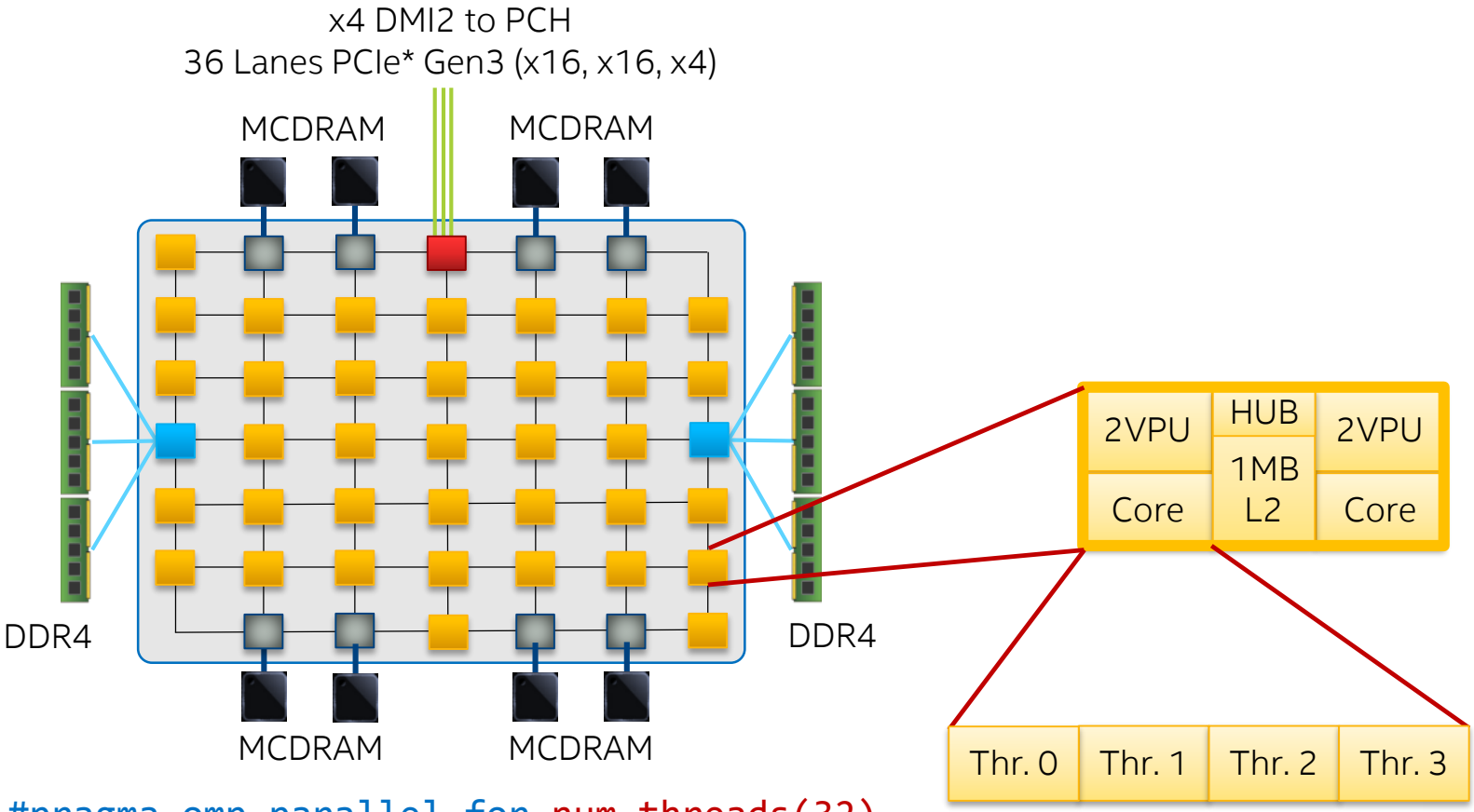
- (Almost) all multi-socket compute servers are NUMA systems
  - Different access latencies for different memory locations
  - Different bandwidth observed for different memory locations
- Example: Intel® Xeon E5-2600v2 Series processor

Thread Affinity – Why It Matters?



SOFTWARE AND SERVICES

Intel Xeon Phi Tile Architecture




```
#pragma omp parallel for num_threads(32)
for (…)
#pragma omp parallel for num_threads(8)
    for (…) {}
```

Thread Affinity – Processor Binding

Binding strategies depends on machine and the app

- Putting threads far, i.e. on different packages
  - (May) improve the aggregated memory bandwidth
  - (May) improve the combined cache size
  - (May) decrease performance of synchronization constructs
- Putting threads close together, i.e. on two adjacent cores which possible share the cache
  - (May) improve performance of synchronization constructs
  - (May) decrease the available memory bandwidth and cache size (per thread)

Thread Affinity in OpenMP* 4.0

- OpenMP 4.0 introduces the concept of places…
  - set of threads running on one or more processors
  - can be defined by the user
  - pre-defined places available:
    - threads one place per hyper-thread
    - cores one place exists per physical core
    - sockets one place per processor package

… and affinity policies…

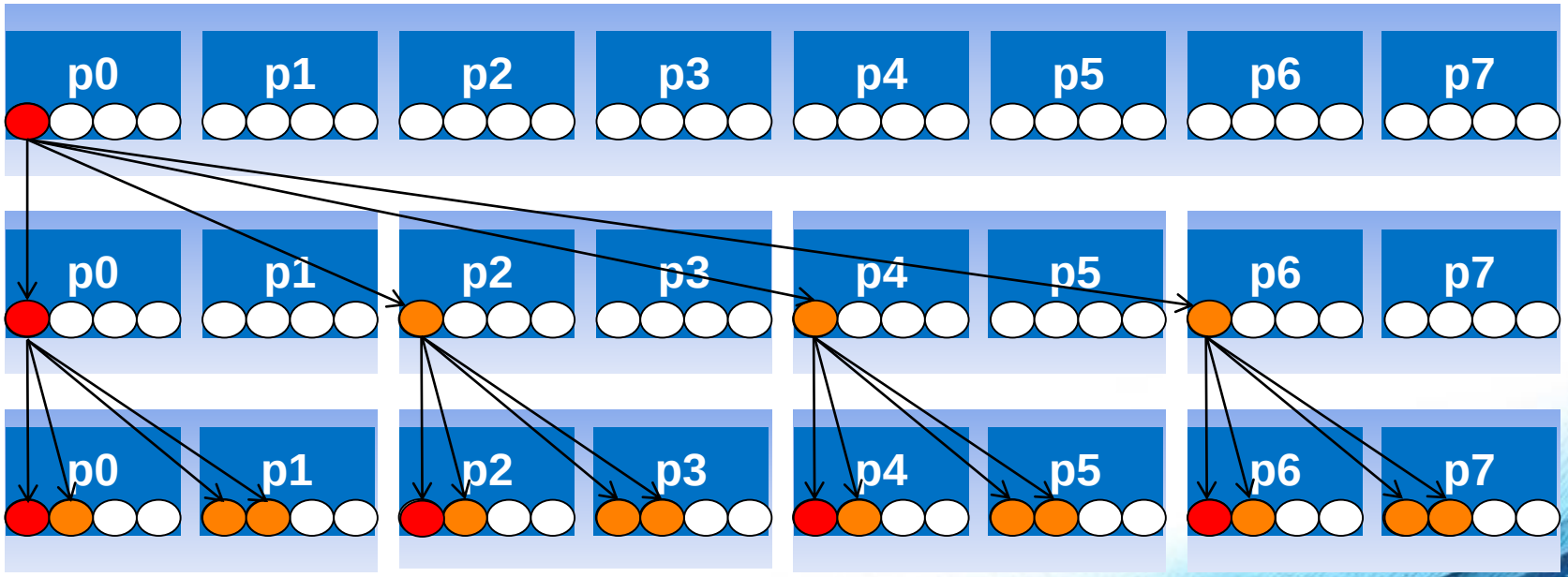
- spread spread OpenMP threads evenly among the places
- close pack OpenMP threads near master thread
- master collocate OpenMP thread with master thread

- … and means to control these settings
  - Environment variables OMP_PLACES and OMP_PROC_BIND
  - clause `proc_bind` for parallel regions

Thread Affinity Example

- Example (Intel® Xeon Phi™ Processor):
  Distribute outer region, keep inner regions close

```
OMP_PLACES=cores(8); OMP_NUM_THREADS=4,4

#pragma omp parallel proc_bind(spread)
   #pragma omp parallel proc_bind(close)
```

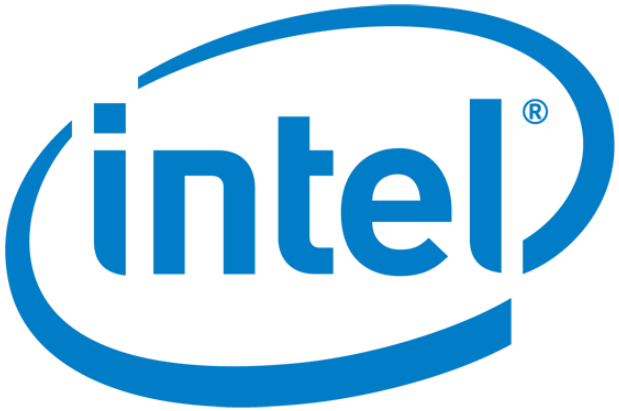


We're Almost Through

- There are so many things in OpenMP today
  - Can't cover all of them in 90 minutes!
- OpenMP 4.0 and 4.5 have more to offer!
  - Improved Fortran 2003 support
  - User-defined reductions
  - Task dependencies
  - Cancellation
  - "doacross" Loops
- We can chat about these features in 1:1s, FTFs, phone calls, or in emails ☺

The last Slide…

- OpenMP 4.5 is not only a bugfix release
  - Task-generating loops
  - Locks with hints
  - Improved support for offloading (if it matters to you)
- Work on OpenMP 5.0 has already been started
  - Expected release during Supercomputing 2018
  - OpenMP 5.0 Beta is scheduled for Supercomputing 2017
  - Features being discussed:
    - Bugfixes ☺
    - Futures
    - Error handling
    - Transactional memory
    - Extensions to tasking
    - Fortran 2008 support
    - C++1x support
    - Data locality and affinity


intel