TAMING INTEL® XEON® PROCESSORS WITH OPENMP*

Dr.-Ing. Michael Klemm

Senior Application Engineer
Developer Relations Division
michael.klemm@intel.com

Chief Executive Officer
OpenMP* Architecture Review Board
michael.klemm@openmp.org

Legal 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.

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.

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

Contents

- Intel Xeon Scalable (Micro-)architecture
- OpenMP Tasking
- OpenMP SIMD
- OpenMP Memory and Thread Affinity

INTEL® XEON® (MICRO-)ARCHITECTURE

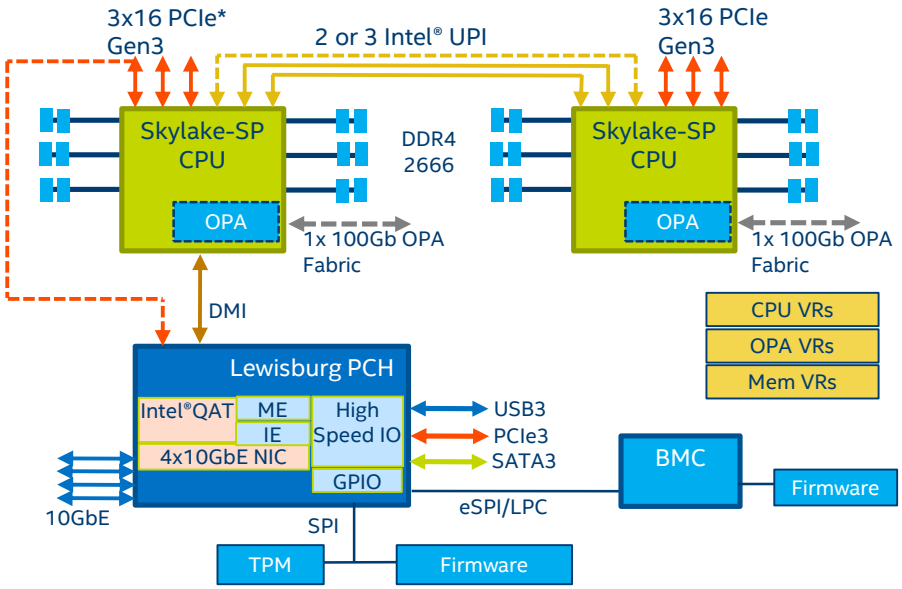
Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

- 512-bit wide vectors
- 32 operand registers
- 8 64b mask registers
- Embedded broadcast
- Embedded rounding

Microarchitecture	Instruction Set	SP FLOPs / cycle	DP FLOPs / cycle
Skylake	Intel® AVX-512 & FMA	64	32
Haswell / Broadwell	Intel AVX2 & FMA	32	16
Sandybridge	Intel AVX (256b)	16	8
Nehalem	SSE (128b)	8	4

Intel AVX-512 Instruction Types	
AVX-512-F	AVX-512 Foundation Instructions
AVX-512-VL	Vector Length Orthogonality : ability to operate on sub-512 vector sizes
AVX-512-BW	512-bit Byte/Word support
AVX-512-DQ	Additional D/Q/SP/DP instructions (converts, transcendental support, etc.)
AVX-512-CD	Conflict Detect : used in vectorizing loops with potential address conflicts

Intel® Xeon® Scalable Processor Node-level Architecture

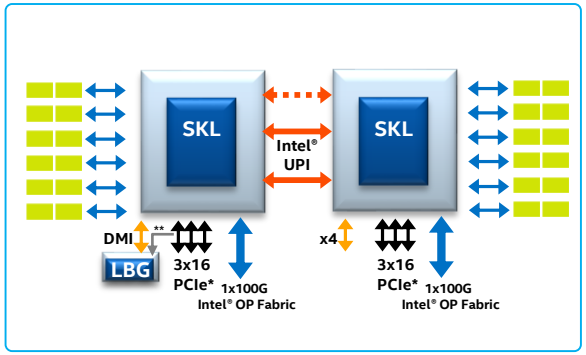


BMC: Baseboard Management Controller	PCH: Intel® Platform Controller Hub	IE: Innovation Engine
Intel® OPA: Intel® Omni-Path Architecture	Intel QAT: Intel® QuickAssist Technology	ME: Manageability Engine
NIC: Network Interface Controller	VMD: Volume Management Device	NTB: Non-Transparent Bridge
UPI: Intel® Ultra Path Interconnect		

Feature	Details
Socket	Socket P
Scalability	2S, 4S, 8S, and >8S (with node controller support)
CPU TDP	70W – 205W
Chipset	Intel® C620 Series (code name Lewisburg)
Networking	Intel® Omni-Path Fabric (integrated or discrete) 4x10GbE (integrated w/ chipset) 100G/40G/25G discrete options
Compression and Crypto Acceleration	Intel® QuickAssist Technology to support 100Gb/s comp/decomp/crypto 100K RSA2K public key
Storage	Integrated QuickData Technology, VMD, and NTB Intel® Optane™ SSD, Intel® 3D-NAND NVMe & SATA SSD
Security	CPU enhancements (MBE, PPK, MPX) Manageability Engine Intel® Platform Trust Technology Intel® Key Protection Technology
Manageability	Innovation Engine (IE) Intel® Node Manager Intel® Datacenter Manager

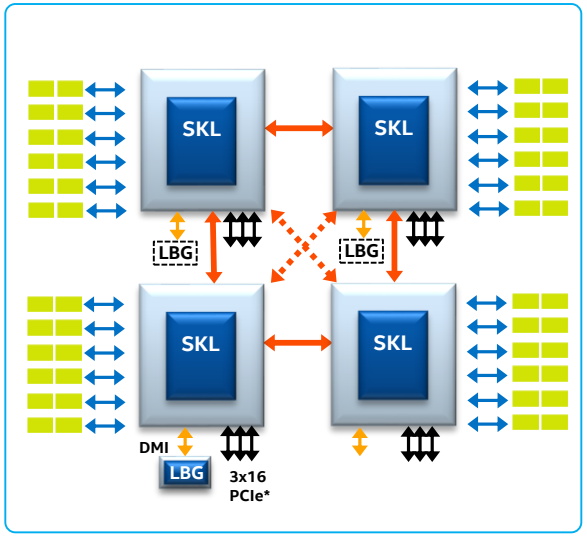
Platform Topologies

2S Configurations



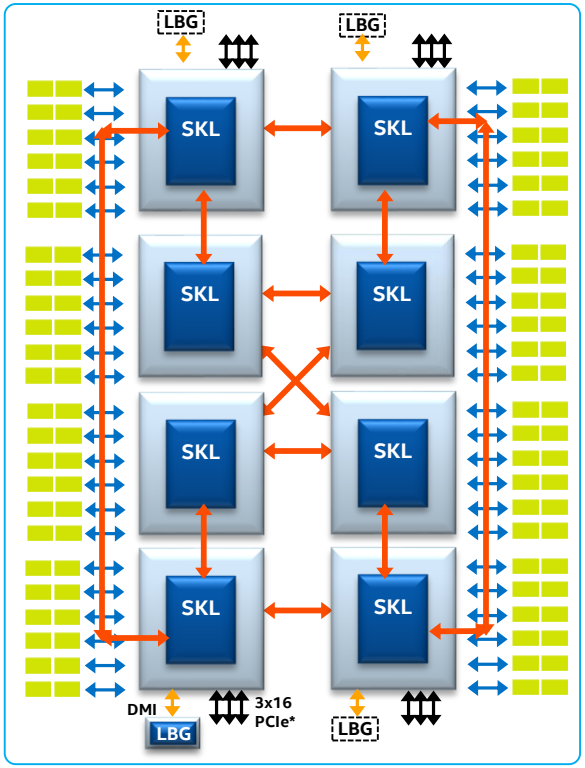
(2S-2UPI & 2S-3UPI shown)

4S Configurations



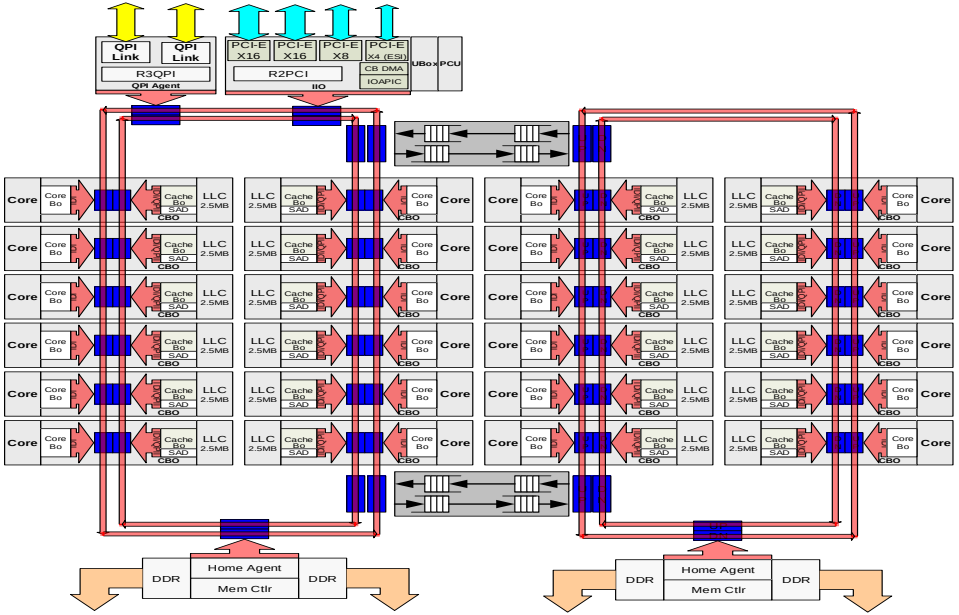
(4S-2UPI & 4S-3UPI shown)

8S Configuration

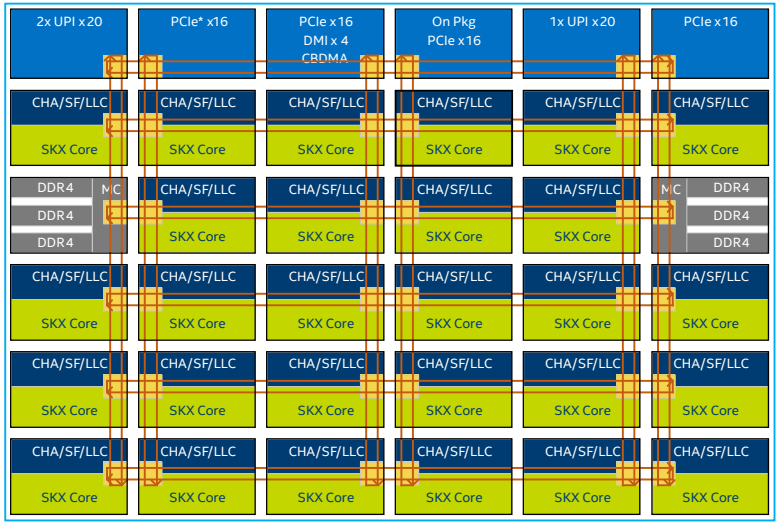


Mesh Interconnect Architecture

Broadwell EX 24-core die

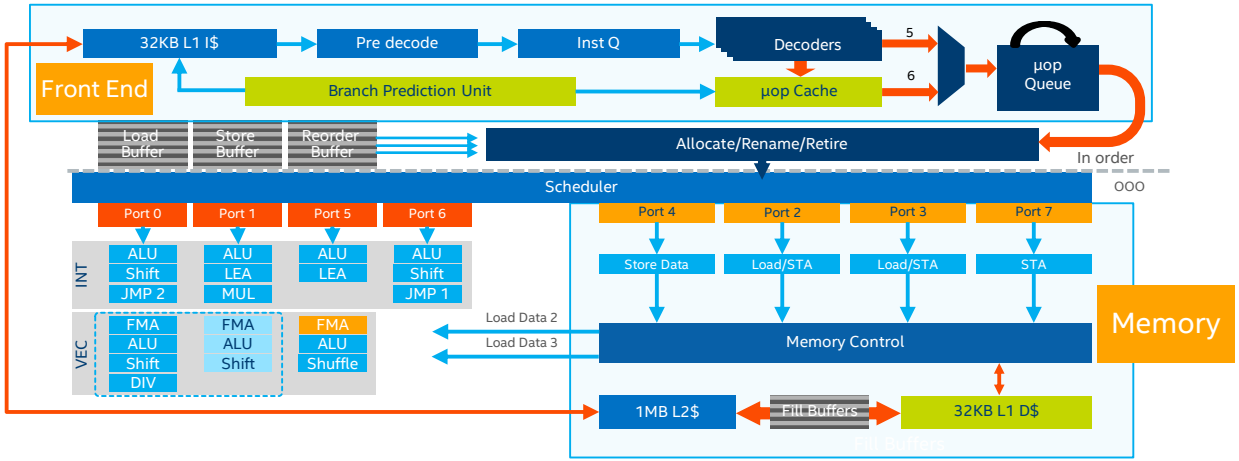


Skylake-SP 28-core die



CHA – Caching and Home Agent ; SF– Snoop Filter; LLC – Last Level Cache ;
SKX Core – Skylake Server Core ; UPI – Intel® UltraPath Interconnect

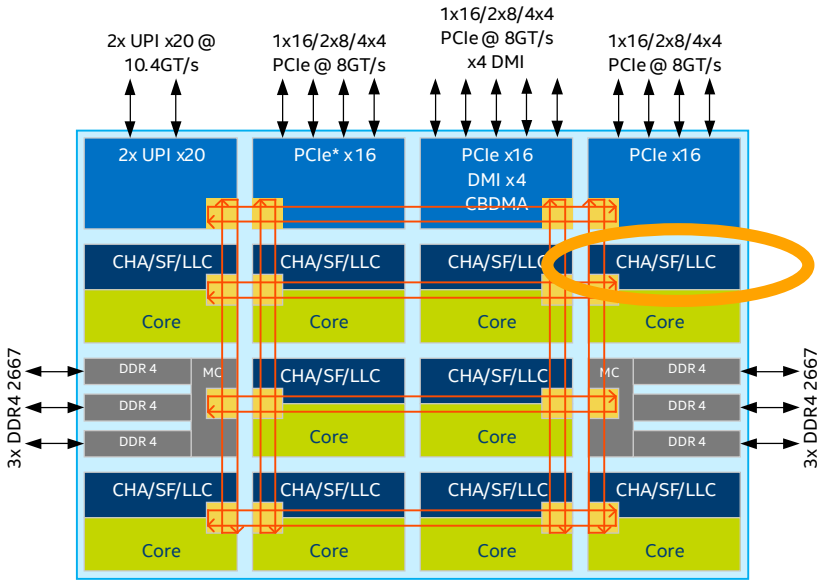
“Skylake” Core Microarchitecture



	Broadwell uArch	Skylake uArch
Out-of-order Window	192	**224**
In-flight Loads + Stores	72 + 42	72 + **56**
Scheduler Entries	60	**97**
Registers – Integer + FP	168 + 168	**180** + 168
Allocation Queue	56	**64/thread**
L1D BW (B/Cyc) – Load + Store	64 + 32	**128 + 64**
L2 Unified TLB	4K+2M: 1024	4K+2M: **1536** **1G: 16**

- Larger and improved branch predictor, higher throughput decoder, larger window to extract ILP
- Improved scheduler and execution engine, improved throughput and latency of divide/sqrt
- More load/store bandwidth, deeper load/store buffers, improved prefetcher

Distributed Caching and Home Agent (CHA)

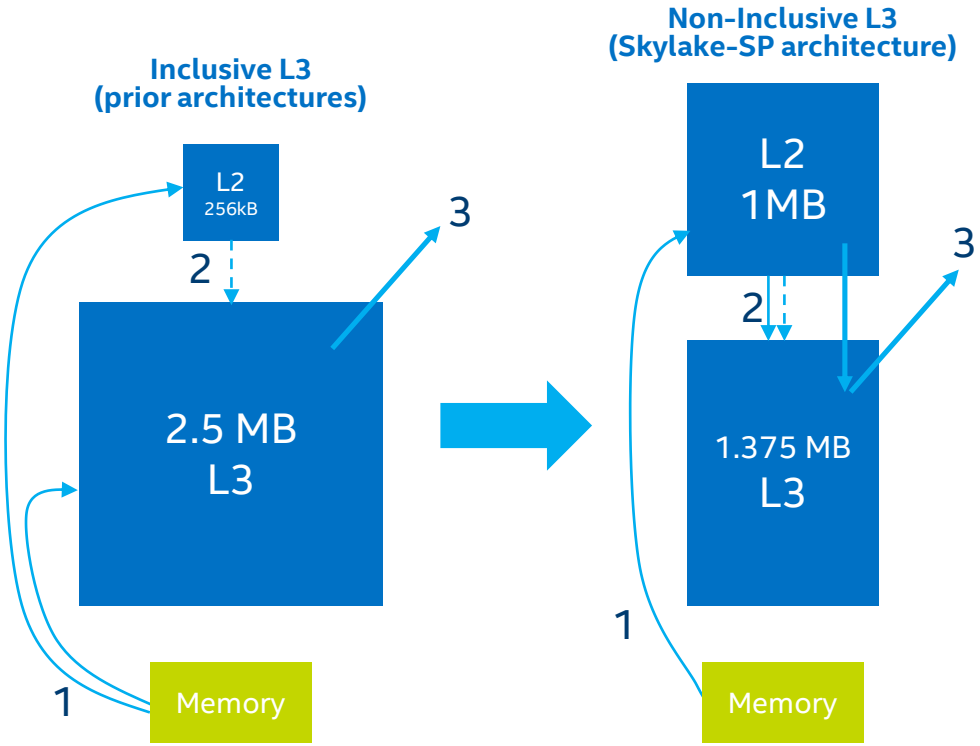


- Intel® UPI caching and home agents are distributed with each LLC bank
  - Prior generation had a small number of QPI home agents
- Distributed CHA benefits
  - Eliminates large tracker structures at memory controllers, allowing more requests in flight and processes them concurrently
  - Reduces traffic on mesh by eliminating home agent to LLC interaction
  - Reduces latency by launching snoops earlier and obviates need for different snoop modes

Re-Architected L2 & L3 Cache Hierarchy

- On-chip cache balance shifted from shared-distributed (prior architectures) to private-local (Skylake architecture):
  - Shared-distributed ➔ shared-distributed L3 is primary cache
  - Private-local ➔ private L2 becomes primary cache with shared L3 used as overflow cache
- Shared L3 changed from inclusive to non-inclusive:
  - Inclusive (prior architectures) ➔ L3 has copies of all lines in L2
  - Non-inclusive (Skylake architecture) ➔ lines in L2 ***may not*** exist in L3

Inclusive vs Non-Inclusive L3

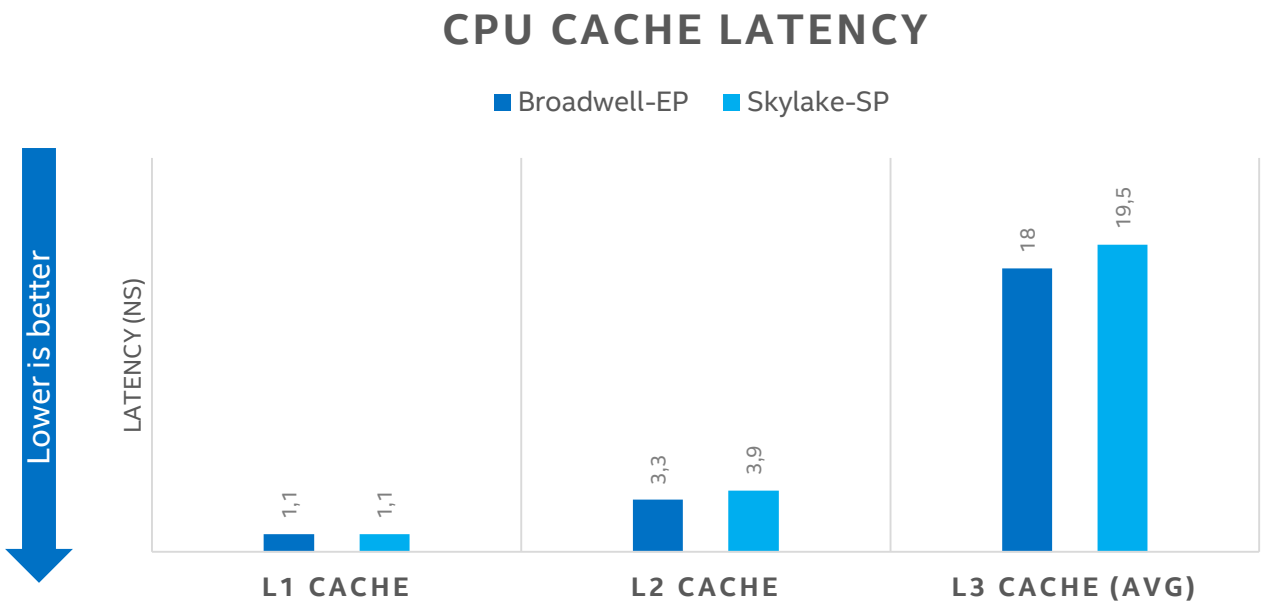


1. Memory reads fill directly to the L2, no longer to both the L2 and L3
2. When a L2 line needs to be removed, both modified and unmodified lines are written back
3. Data shared across cores are copied into the L3 for servicing future L2 misses

Cache hierarchy architected and optimized for data center use cases:

- Virtualized use cases get larger private L2 cache free from interference
- Multithreaded workloads can operate on larger data per thread (due to increased L2 size) and reduce uncore activity

Cache Performance



Skylake-SP L2 cache latency has increased by 2 cycles for a 4x larger L2

Skylake-SP achieves good L3 cache latency even with larger core count

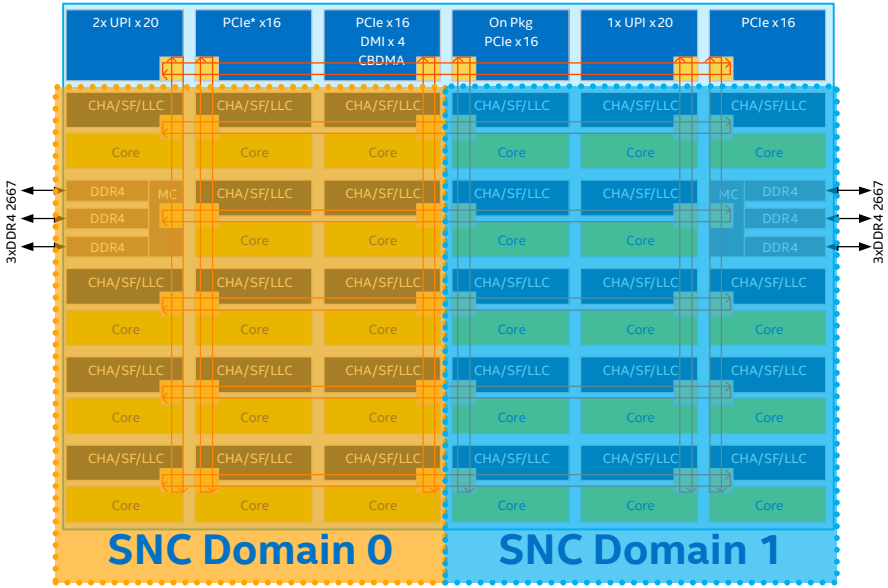
Source as of June 2017: Intel internal measurements on platform with Xeon Platinum 8180, Turbo enabled, SNC1, 6x32GB DDR4-2666 per CPU, 1 DPC, and platform with Intel® Xeon® E5-2699 v4, Turbo enabled, without COD, 4x32GB DDR4-2400, RHEL 7.0. Cache latency measurements were done using Intel® Memory Latency Checker (MLC) tool.
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/performance.

Sub-NUMA Cluster (SNC)

Prior generation supported Cluster-On-Die (COD)

SNC provides similar localization benefits as COD, without some of its downsides

- Only one UPI caching agent required even in 2-SNC mode
- Latency for memory accesses in remote cluster is smaller, no UPI flow
- LLC capacity is utilized more efficiently in 2-cluster mode, no duplication of lines in LLC

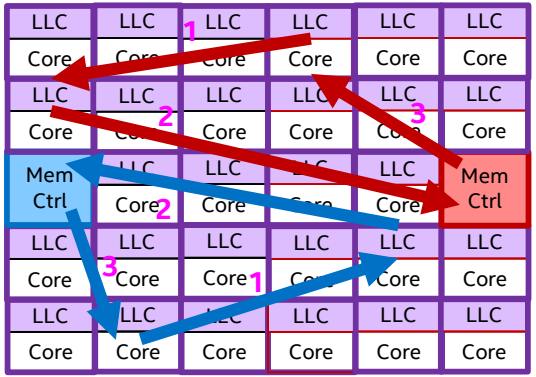


Sub-NUMA Clusters – 2 SNC Example

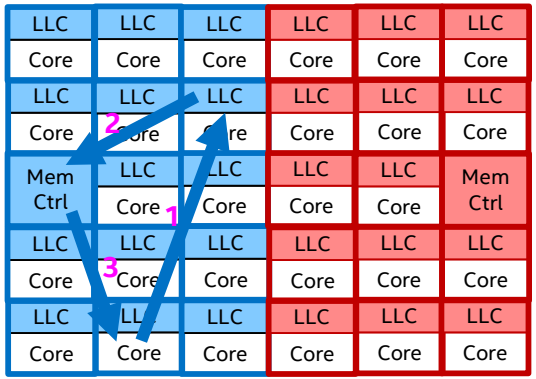
SNC partitions the LLC banks and associates them with memory controller to localize LLC miss traffic

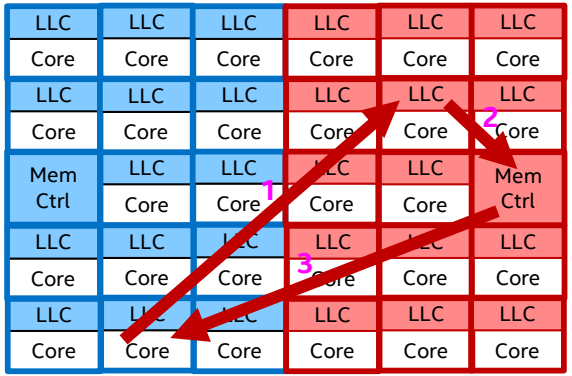
- LLC miss latency to local cluster is smaller
- Mesh traffic is localized, reducing uncore power and sustaining higher BW

Without SNC



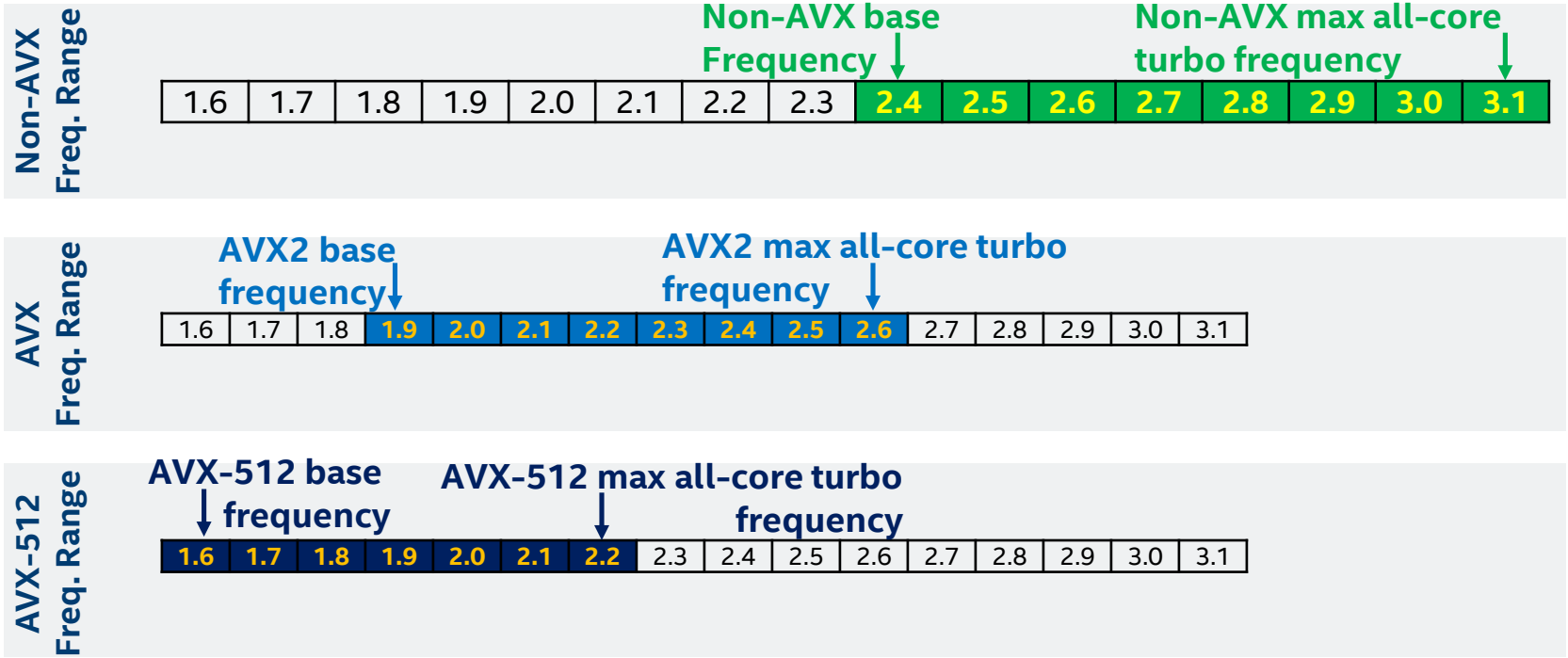
Local SNC Access





Remote SNC Access

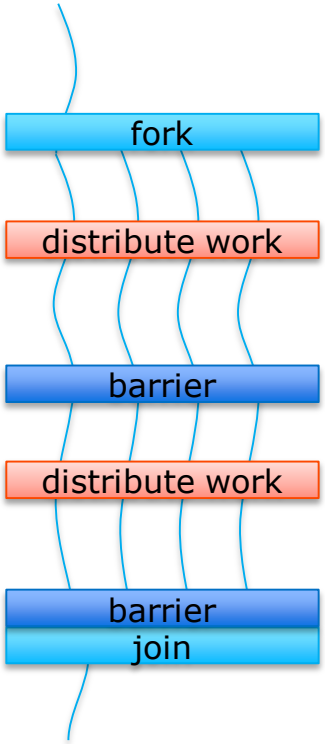
AVX Frequency – All Core Turbo



OPENMP* TASKING

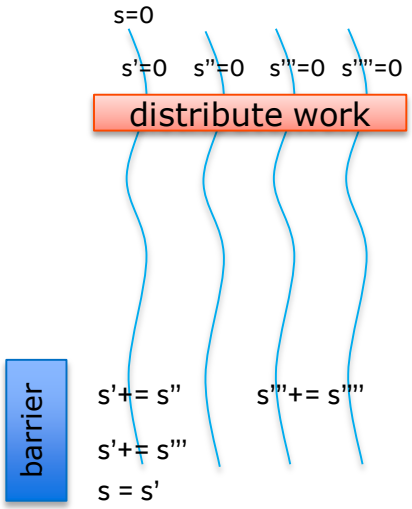
OpenMP Worksharing

```
#pragma omp parallel
{

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



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



OpenMP Worksharing/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;


}
```



Traditional Worksharing

Worksharing constructs do not compose well *(or at least: do not compose as well as we want)*
Pathological example: parallel daxpy in MKL

```
void example1() {
    #pragma omp parallel
    {
        compute_in_parallel_this(A); // for, sects,…
        compute_in_parallel_that(B); // for, sects,…
        // daxpy is either parallel or sequential,
        // but has no orphaned worksharing
        cblas_daxpy (n, x, A, incx, B, incy);
    }
}
```

```
void example2() {

    // parallel within: this/that
    compute_in_parallel_this(A);
    compute_in_parallel_that(B);

    // parallel MKL version
    cblas_daxpy ( <...> );

}
```

Writing such codes either:

- oversubscribes the system (creating more OpenMP threads than cores)
- yields bad performance due to OpenMP overheads, or
- needs a lot of glue code to use sequential daxpy only for sub-arrays

Task Execution Model

Supports unstructured parallelism

- unbounded loops

```
while ( <expr> ) {
   ...
}
```

- recursive functions

```
void myfunc( <args> )
{
   ...; myfunc( <newargs> ); ...;
}
```

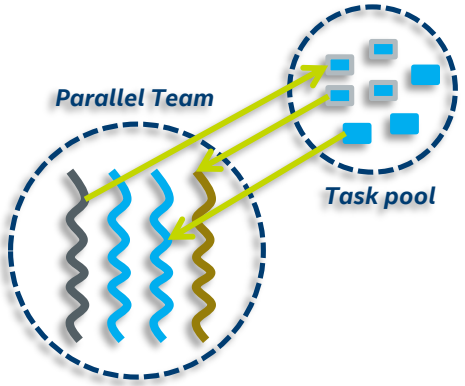
Several scenarios are possible:

- single creator, multiple creators, nested tasks (tasks & WS)

All threads in the team are candidates to execute tasks

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```



The task Construct

Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

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

Data Environment

- private(list)
- firstprivate(list)
- shared(list)
- default(shared | none)
- *in_reduction(r-id: list)**

Miscellaneous

- *allocate([allocator:] list)**
- *detach(event-handler)**

Cutoff Strategies

- if(scalar-expression)
- mergeable
- final(scalar-expression)

Dependencies

- depend(dep-type: list)

Sched. Restriction

- untied

Scheduler Hints

- priority(priority-value)
- *affinity(list)**

Task Synchronization

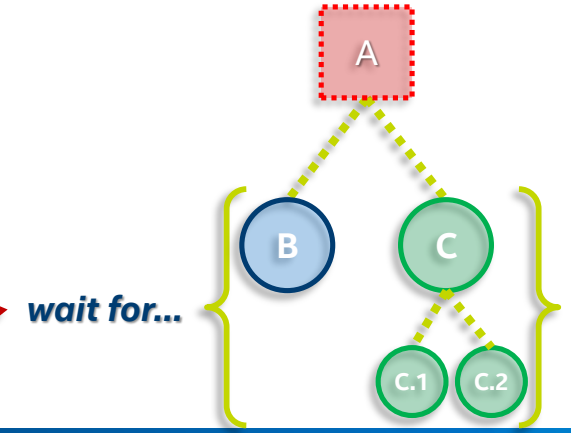
The taskgroup construct (deep task synchronization)

- attached to a structured block; completion of all descendants of the current task; TSP at the end

```
#pragma omp taskgroup [clause[[,] clause]...]
{structured-block}
```

- where clause (could only be): reduction(reduction-identifier: list-items) ≥ OpenMP 5.0

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp taskgroup
   {
      #pragma omp task
      { … }
      #pragma omp task
      { … #C.1; #C.2; …}

   } // end of taskgroup
}
```



Tasking Use Case: Cholesky Factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```

Complex synchronization patterns

- Splitting computational phases
- taskwait or taskgroup
- Needs complex code analysis
- May perform a bit better than regular OpenMP worksharing

Task Reductions (using taskgroup)

Reduction operation

- perform some forms of recurrence calculations
- associative and commutative operators

The (taskgroup) scoping reduction clause

```
#pragma omp taskgroup task_reduction(op: list)
{structured-block}
```

- Register a new reduction at [1]

```
#pragma omp task in_reduction(op: list)
{structured-block}
```

- Task participates in a reduction operation [2]

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \
                firstprivate(node)
        { // [2]
          res += node->value;
        }
        node = node->next;
      }
    } // [3]
  }
}
```

OpenMP 5.0

Tasking Use Case: parallel saxpy

```
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

```
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

```
#pragma omp parallel
#pragma omp single
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   #pragma omp task private(ii) \
           firstprivate(i,UB) shared(S,A,B)
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

Difficult to determine grain

- 1 single iteration → to fine
- whole loop → no parallelism

Manually transform the code

- blocking techniques

Improving programmability

- OpenMP taskloop

Example: saxpy Kernel with OpenMP taskloop

```
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

blocking

```
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

```
for ( i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS)?SIZE:i+TS;
   #pragma omp task private(ii) \
    firstprivate(i,UB) shared(S,A,B)
   for ( ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

taskloop

```
#pragma omp taskloop grainsize(TS)
for ( i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

Easier to apply than manual blocking:

- Compiler implements mechanical transformation
- Less error-prone, more productive

Worksharing vs. taskloop Constructs (1/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 16384

Worksharing vs. taskloop Constructs (2/2)

```
subroutine worksharing
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)

!$omp do
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end do

!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

```
subroutine taskloop
    integer :: x
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x = 0
!$omp parallel shared(x) num_threads(T)
!$omp single
!$omp taskloop
    do i = 1,N
!$omp atomic
        x = x + 1
!$omp end atomic
    end do
!$omp end taskloop
!$omp end single
!$omp end parallel
    write (*,'(A,I0)') 'x = ', x
end subroutine
```

Result: x = 1024

Tasking Use Case: Cholesky Factorization

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    potrf(a[k][k], ts, ts);
    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait
    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```

Complex synchronization patterns

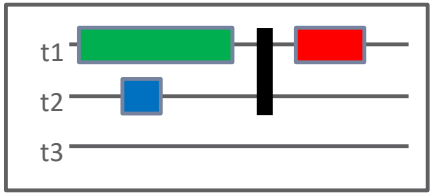
- Splitting computational phases
- taskwait or taskgroup
- Needs complex code analysis
- May perform a bit better than regular OpenMP worksharing

Is this best solution we can come up with?

Task Synchronization w/ Dependencies

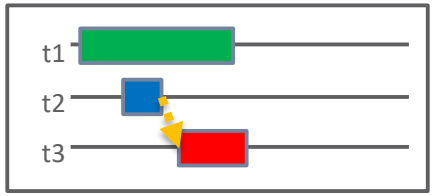
OpenMP 3.1

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
#pragma omp task
  std::cout << x << std::endl;

#pragma omp task
  long_running_task();

  #pragma omp taskwait

#pragma omp task
  x++;
}
```



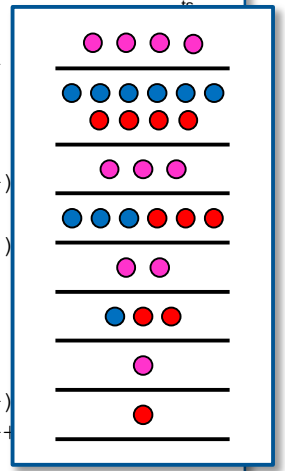
OpenMP 4.0

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(in: x)
  std::cout << x << std::endl;

#pragma omp task
  long_running_task();

#pragma omp task depend(inout: x)
  x++;
}
```



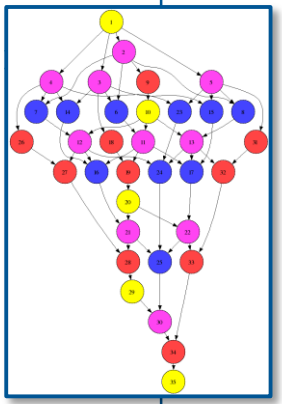
Example: Cholesky Factorization

```
void cholesky(int ts, int nt, double* a[nt][nt])
{
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++)
    #pragma omp task
      trsm(a[k][k], a[k][i], ts, ts);
    }
    #pragma omp taskwait

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++)
      for (int j = k + 1; j < i; j++
      #pragma omp task
        dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
    #pragma omp task
      syrk(a[k][i], a[i][i], ts, ts);
    }
    #pragma omp taskwait
  }
}
```



OpenMP 3.1

```
void cholesky(int ts, int nt, double* a[nt][nt]) {
  for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    #pragma omp task depend(inout: a[k][k]
    potrf(a[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend(in: a[k][k])
                       depend(inout: a[k][i])
    trsm(a[k][k], a[k][i], ts, ts);
    }

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
        #pragma omp task depend(inout: a[j][i])
                         depend(in: a[k][i], a[k][j])
      dgemm(a[k][i], a[k][j], a[j][i], ts, ts);
      }
      #pragma omp task depend(inout: a[i][i])
                       depend(in: a[k][i])
    syrk(a[k][i], a[i][i], ts, ts);
    }
  }
}
```

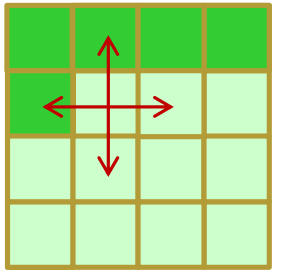


OpenMP 4.0

Use Case: Gauss-Seidel Stencil Code (1/5)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern

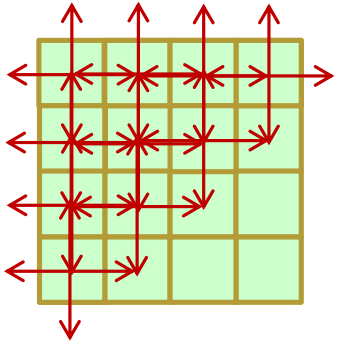


- Dependence
  - Two cells from the current time step (N & W)
  - Two cells from the previous time step (S & E)

Use Case: Gauss-Seidel Stencil Code (2/5)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern



- Dependence
  - Two cells from the current time step (N & W)
  - Two cells from the previous time step (S & E)

Use Case: Gauss-Seidel Stencil Code (3/5)

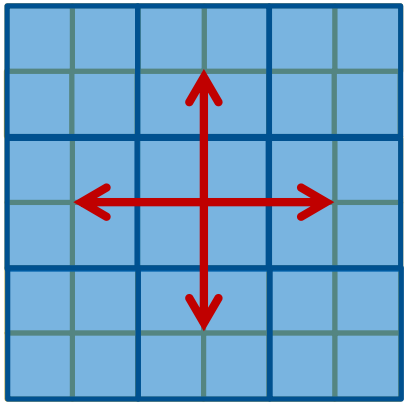
```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
           for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
             for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
             p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                               p[i-1][j] * p[i+1][j]);
    } }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
} } }
```

Works, but

- creates ragged fork/join,
- makes excessive use of barriers, and
- overly limits parallelism.

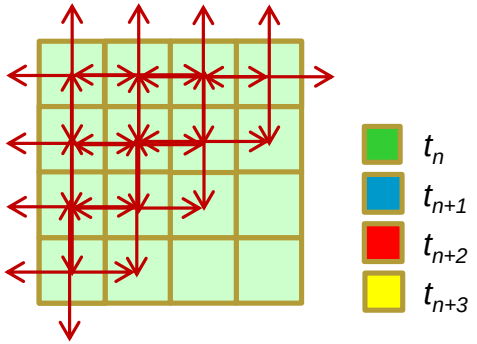
Use Case: Gauss-Seidel Stencil Code (4/5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                 p[i-1][j] * p[i+1][j]);
        }
      }
}
```



Use Case: Gauss-Seidel Stencil Code (4/5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
               p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                 p[i-1][j] * p[i+1][j]);
        }
      }
}
```



OPENMP* SIMD PROGRAMMING

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++)

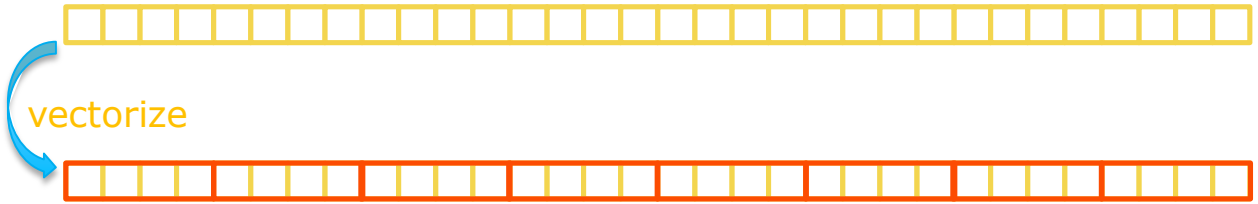
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

Syntax (Fortran)

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

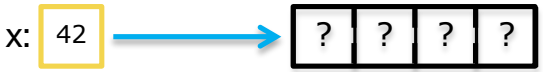
Example

```
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's value is in relationship with the iteration number
  - $x_i = x_{orig} + i * \text{linear-step}$

`aligned (list[:alignment])`

- Specifies that the list items have a given alignment
- Default is alignment for the architecture

`collapse (n)`

SIMD Worksharing Construct

Parallelize and vectorize a loop nest

- Distribute a loop's iteration space across a thread team
- Subdivide loop chunks to fit a SIMD vector register

Syntax (C/C++)

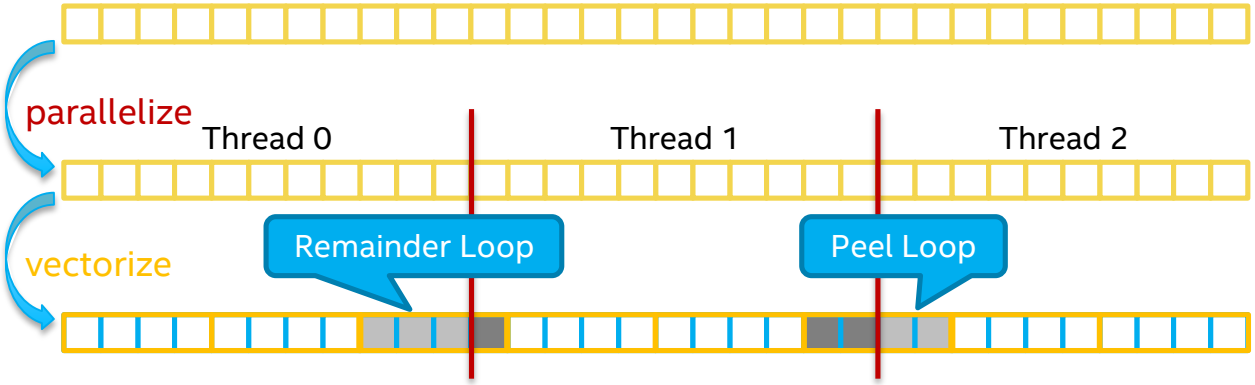
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

Syntax (Fortran)

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

Example

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



Be Careful What You Wish For…

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

You should choose chunk sizes that are multiples of the SIMD length

- Remainder loops are not triggered
- Likely better performance

In the above example …

- and AVX2 (= 8-wide), the code will only execute the remainder loop!
- and SSE (=4-wide), the code will have one iteration in the SIMD loop plus one in the remainder loop!

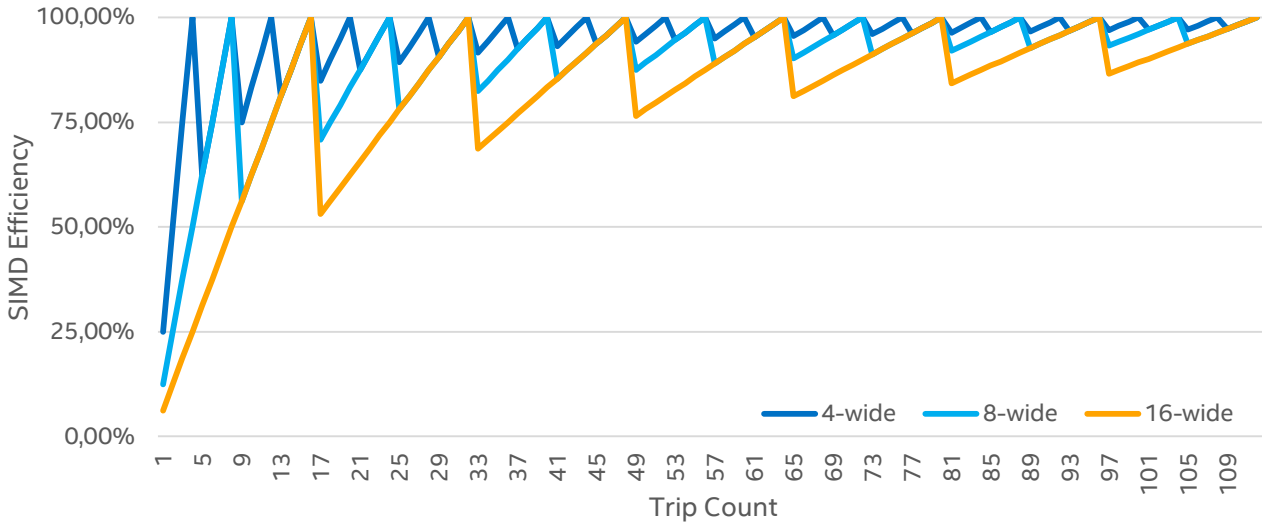
Vectorization Efficiency

Vectorization efficiency is a measure how well the code uses SIMD features

- Corresponds to the average utilization of SIMD registers for a loop
- Defined as (*N*: trip count, *vl*: vector length): $VE = \frac{N/vl}{\lceil N/vl \rceil}$

For 8-wide SIMD:

- *N* = 1: 12.50%
- *N* = 2: 25.00%
- *N* = 4: 50.00%
- *N* = 8: 100.00%
- *N* = 9: 56.25%
- *N* = 16: 100.00%



OpenMP 4.5 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]);
}   }
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):
    vminps %zmm1, %zmm0, %zmm0
    ret
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):
    vsubps %zmm0, %zmm1, %zmm2
    vmulps %zmm2, %zmm2, %zmm0
    ret
```

```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

AT&T syntax: destination operand is on the right

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`

- optimize for 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]`*`)`

MEMORY AND THREAD AFFINITY

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

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, cores, sockets

… and affinity policies…

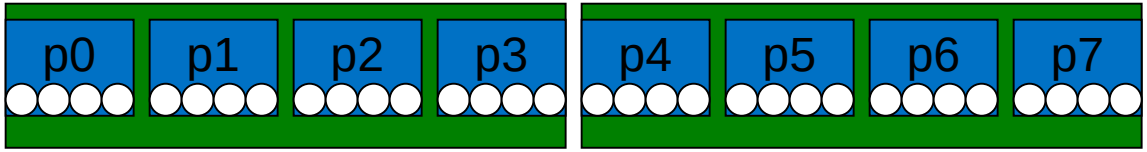
- spread, close, master

… and means to control these settings

- Environment variables OMP_PLACES and OMP_PROC_BIND
- clause proc_bind for parallel regions

OpenMP Places

Imagine this machine:



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

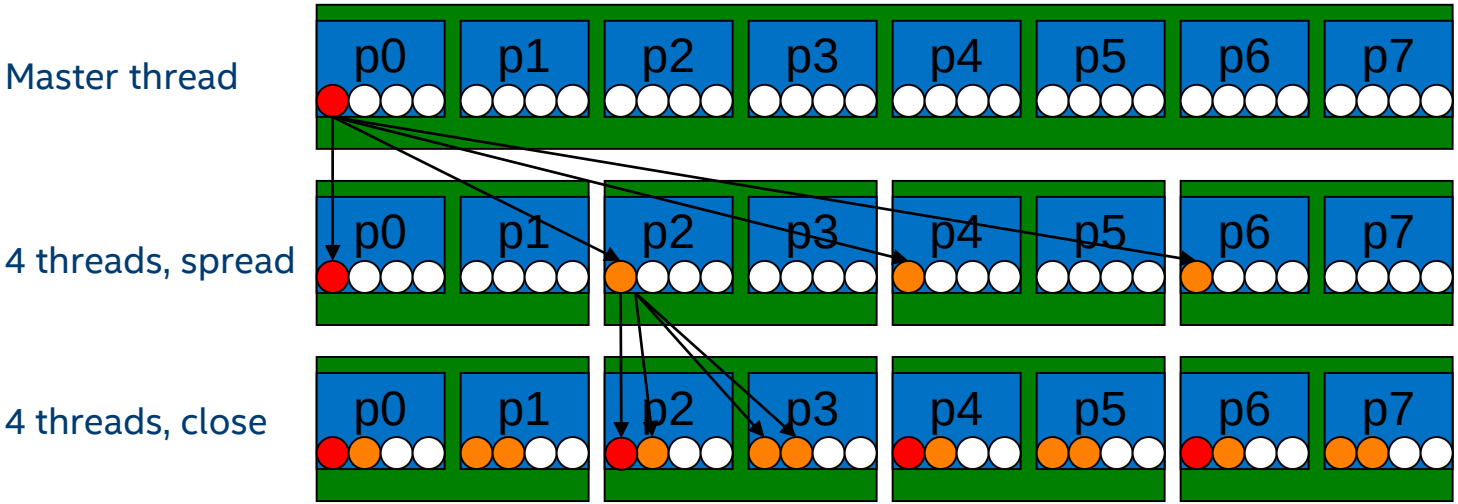
Abstract names for OMP_PLACES:

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

OpenMP Places and Policies

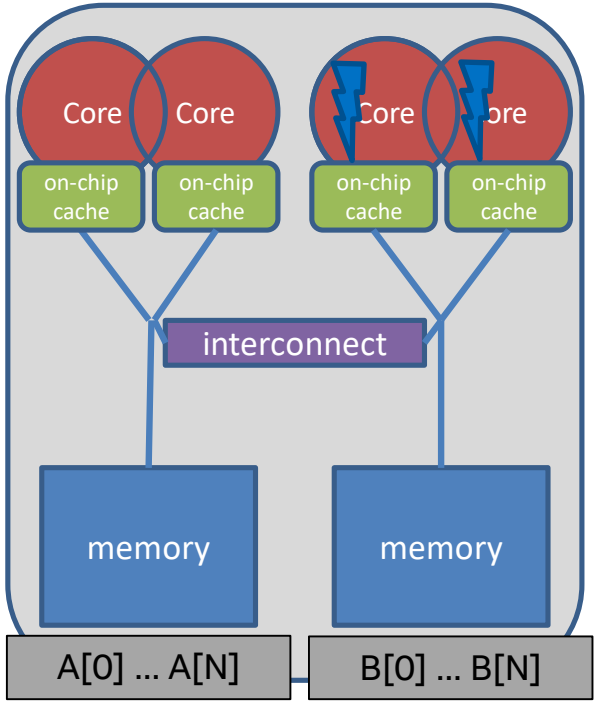
Example: separate cores for outer loop and near cores for inner loop

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



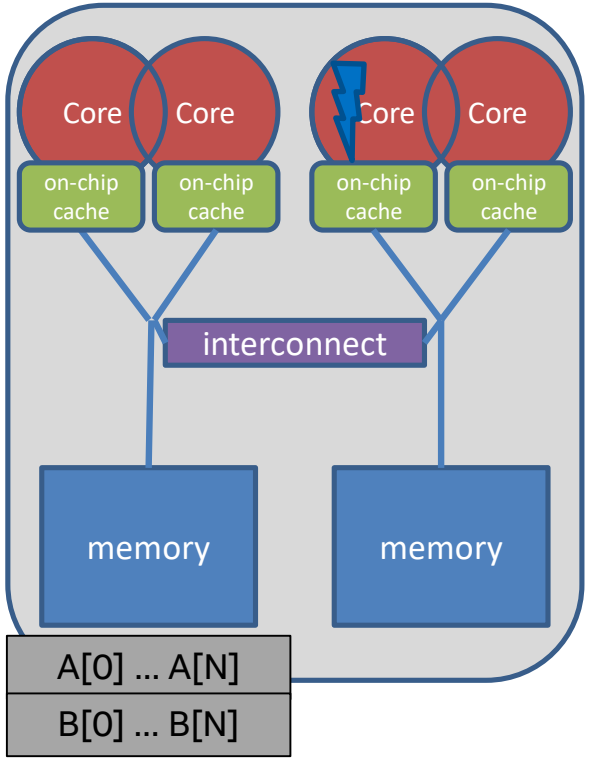
OpenMP Task Affinity

```
void task_affinity() {
    double* B;
#pragma omp task shared(B)
    {
        B = init_B_and_important_computation(A);
    }
#pragma omp task firstprivate(B)
    {
        important_computation_too(B);
    }
#pragma omp taskwait
}
```



OpenMP Task Affinity

```
void task_affinity() {
    double* B;
#pragma omp task shared(B) affinity(A[0:N])
    {
        B = init_B_and_important_computation(A);
    }
#pragma omp task firstprivate(B) affinity(B[0:N])
    {
        important_computation_too(B);
    }
#pragma omp taskwait
}
```



User Control of Memory Placement

Explicit NUMA-aware memory allocation:

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

Example: using numactl to distribute pages round-robin:

- `numactl –interleave=all ./a.out`

Memory Allocators (OpenMP API v5.0)

New clause on all constructs with data sharing clauses:

- `allocate( [allocator:] list )`

Allocation:

- `omp_alloc(size_t size, omp_allocator_t *allocator)`

Deallocation:

- `omp_free(void *ptr, const omp_allocator_t *allocator)`
- `allocator` argument is optional

`allocate` directive

- Standalone directive for allocation, or declaration of allocation statement

Example: Using Memory Allocators (v5.0)

```
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR and/or omp_set_default_allocator
    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c) // on target; must be compile-time expr
    {
         #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
      {
            some_other_parallel_code();
       }
    }

    omp_free(p);
}
```

Partitioning Memory w/ OpenMP version 5.0

```
void allocator_example() {
    double *array;

    omp_allocator_t *allocator;
    omp_alloctrait_t traits[] = {
        {OMP_ATK_PARTITION, OMP_ATV_BLOCKED}
    };
    int ntraits = sizeof(traits) / sizeof(*traits);
    allocator = omp_init_allocator(omp_default_mem_space, ntraits, traits);

    array = omp_alloc(sizeof(*array) * N, allocator);

#pragma omp parallel for proc_bind(spread)
    for (int i = 0; i < N; ++i) {
        important_computation(&array[i]);
    }

    omp_free(array);
}
```

ALMOST AT THE END...

Advert: OpenMPCon and IWOMP 2018

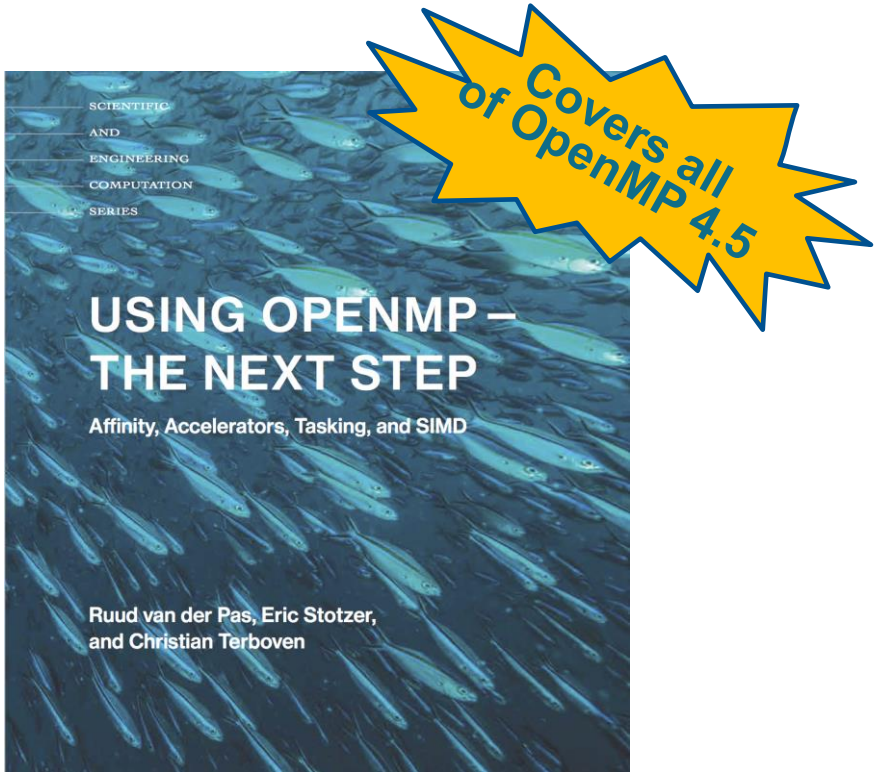
Conference dates:

- OpenMPCon: Sep 24-25
- Tutorials: Sep 26
- IWOMP: Sep 27-28

Co-located with EuroMPI

Location: Barcelona, Spain (?)

Advert: OpenMP Book



OpenMP v5.0 is on its Way (Release @ SC18)

Detachable Tasks

Memory Allocators

Task Reductions

Tools APIs

Dependence Objects

C++14 and C++17 support

Unified Shared Memory

`loop` Construct

Fortran 2008 support

Collapse non-rect. Loops

Task-to-data Affinity

Multi-level Parallelism

Data Serialization for Offload

Meta-directives

“Reverse Offloading”

Parallel Scan

Improved Task Dependences

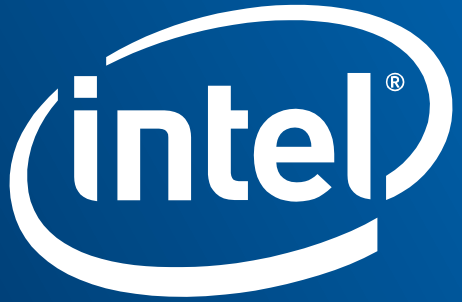
Summary

Modern high-performance processors are massively parallel processors

- Multi-core/many-core
- SIMD execution

OpenMP offers powerful mechanisms to program massively parallel processors

- Tasking incl. data-driven task dependences
- SIMD directives to guide compiler to emit data-parallel instructions
- Features to control memory and thread affinity


intel