OpenMP* Threading

Dr. Mikko Byckling
Senior Application Engineer

Acknowledgements: Martyn Corden, Intel; Steve “Dr. Fortran” Lionel, ex-Intel

*Other names and brands may be claimed as the property of others.

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

All products, computer systems, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel processors of the same SKU may vary in frequency or power as a result of natural variability in the production process.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

The benchmark results reported above may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

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 www.intel.com/benchmarks.

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.

No computer system can be absolutely secure.

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel® Hyper-Threading Technology available on select Intel® processors. Requires an Intel® HT Technology-enabled system. Your performance varies depending on the specific hardware and software you use. Learn more by visiting http://www.intel.com/info/hyperthreading.

© 2018 Intel Corporation. Intel, the Intel logo, Xeon and Xeon logos are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- OpenMP* Threading Model
  - **`parallel`**, **`do`** and **`task`** constructs, **`barrier`**, data scoping, scheduling
- OpenMP* 4.0 Tasking
  - Task **`depend`**
  - OpenMP* 4.5 Tasking: **`taskloop`**
- OpenMP* performance aspects
  - Thread affinity
  - Non-uniform memory access

Copyright © 2018 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Contents

- OpenMP* Threading Model
  - **`parallel`**, **`do`** and **`task`** constructs, **`barrier`**, data scoping, scheduling
- OpenMP* 4.0 Tasking
  - Task **`depend`**
  - OpenMP* 4.5 Tasking: **`taskloop`**
- OpenMP* performance aspects
  - Thread affinity
  - Non-uniform memory access

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

OpenMP Threading Model

- In OpenMP, the description of ***parallelism*** and ***worksharing*** treated as separate entities
- Thread *parallelism* is described with `parallel` construct

  C/C++: `#pragma omp parallel [clauses]`

  Fortran: `!$omp parallel [clauses]`
- Loop *worksharing* is described with `loop` construct

  C/C++: `#pragma omp for [clauses]`

  Fortran: `!$omp do [clauses]`
- Task *worksharing* is described with `task` construct

  C/C++: `#pragma omp task [clauses]`

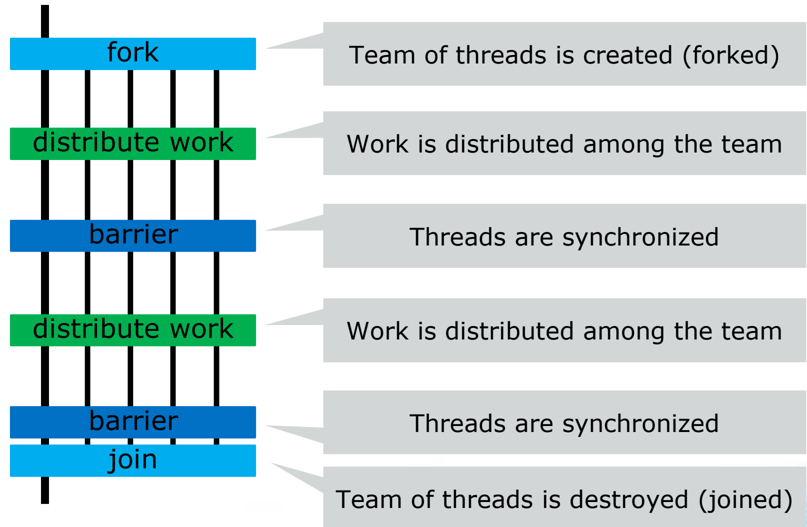
  Fortran: `!$omp task [clauses]`

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel`: Example

```
!$omp parallel
    !$omp do
    DO i=1,N
    …
    END DO
    !$omp end do
    !$omp do
    DO i=1,N
    …
    END DO
    !$omp end do
!$omp end parallel
```

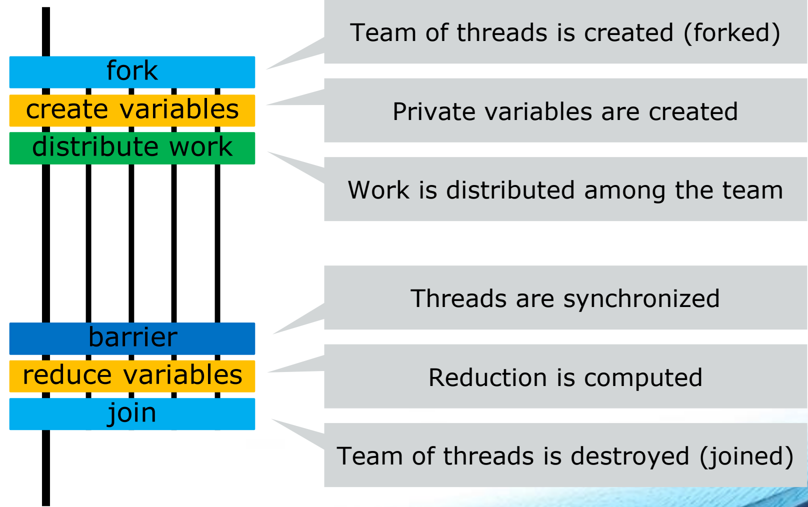


SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel do`: Example

```fortran
REAL :: a(N), l, s
s = 0.0
!$omp parallel do      &
  reduction(+:s)       &
  private(l)           &
  schedule(static,4)
  DO i=1,N
    l = sqrt(a(i))
    s = s + l
  END DO
!$omp end parallel do
```



SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `barrier` construct

- Specify an explicit barrier at the point at which the construct appears
  `!$omp barrier`
  - All threads executing the enclosing parallel region must execute the barrier before any are allowed to continue

```
INTEGER :: n
REAL :: global_data(n,n)
REAL, ALLOCATABLE :: private_data(:,:)
!$omp parallel private(private_data)
  CALL initialize(global_data, private_data)
  !$omp barrier
  CALL compute(data)
!$omp end parallel
```

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `single` construct

- Execute a structured block with any single thread
  - Other threads wait at end of the construct

```
INTEGER :: n
REAL, ALLOCATABLE :: global_data(:,:), global_data_in(:,:)
REAL, ALLOCATABLE :: private_data(:,:)
!$omp parallel private(private_data)
  ! Initialize global data (in parallel)
  CALL initialize_global(global_data)
  !$omp single
  ! Read from file (serial)
  CALL initialize_global_in(global_data_in)
  !$omp end single
  CALL initialize_private(global_data, global_data_in, private_data)
  …
!$omp end parallel
```

Software and Services

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `threadprivate` directive

- Declarative directive
  `!$omp threadprivate(list)`
  - Specify that variables in list are replicated, with each thread having its own copy

```
MODULE thread_data
  REAL, ALLOCATABLE :: data(:,:)
  !$omp threadprivate(data)
CONTAINS
  SUBROUTINE initialize_thread_private(global_data)
    REAL, CONTIGUOUS :: global_data(:,:)
    !$omp parallel
    ALLOCATE(data, SOURCE=global_data)
    !$omp end parallel
  END SUBROUTINE
! Rest of implementation omitted
```

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel` clauses

- `if(expression)`
  - When `expression` evaluates to `.TRUE.`, execute the construct in parallel
- `num_threads(num-threads)`
  - Execute the construct with `num-threads`
- `reduction(identifier:list)`
  - For variables in `list`, make a private copy and compute a reduction with `identifier` after the end of the region
- `proc_bind(master | close | spread)`
  - Execute region with the defined thread affinity

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel` clauses

- **`private(list)`**
  - Declare variables in **`list`** to be private to a thread
  - Implicitly private: variables in a private scope (a called function or **`BLOCK`** construct) within a parallel region
- **`firstprivate(list)`**
  - As private, but initialize the value according to the corresponding original item

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel` clauses

- **`shared(list)`**
  - Declare variable in **`list`** to be shared between threads
  - Implicitly shared: all variables with the **`SAVE`** attribute (incl. implicit SAVE), **`COMMON`** block and **`MODULE`** scope variables
- **`copyin(list)`**
  - Initialize a **`threadprivate`** variable to the value of the master thread

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `parallel` clauses

- `default( firstprivate | none | private | shared )`
  - Set the data sharing attribute for variables which the data scoping would be otherwise implicitly defined
  - NOTE: **`none`** will require each variable referred in the scope of the parallel section to have its data sharing attribute defined

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP loop construct clauses

- `private, firstprivate, reduction`
  - As with parallel
- `lastprivate(list)`
  - As private, but set the value **after** the region according corresponding to the lexically last iteration index

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP loop construct clauses

- `collapse(n)`
  - Combine the iteration space of **n** subsequent tightly nested loops together
- `ordered[(n)]`
  - Execute **n** subsequent tightly nested loops in a sequential order
- `nowait`
  - No implicit barrier at the end of the construct

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP loop construct scheduling

- **`schedule(kind[, chunk_size])`**
  - Distribute work in chunks of **`chunk_size`** to threads with schedule kind, where **`kind`** is
    `static`: distribute chunks to threads in a round-robin fashion
    `dynamic`: chunks are distributed to threads dynamically
    `guided`: as dynamic, but decrease the **`chunk_size`** towards the end of the iteration
    `runtime`: choose schedule at run time (**`OMP_SCHEDULE`**)
  - Default **`chunk_size`** depends on the used schedule **`kind`**

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- OpenMP* Threading Model
  - **parallel**, **do** and **task** constructs, **barrier**, data scoping, scheduling
- OpenMP* 4.0 Tasking
  - Task **depend**
  - OpenMP* 4.5 Tasking: **taskloop**
- OpenMP* performance aspects
  - Thread affinity
  - Non-uniform memory access

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

OpenMP 4.0 Tasking

- OpenMP 3.0 introduced dynamic tasking in addition to static tasking (cf., `sections` construct)
- Implement irregular patterns through tasking
  - Recursion
  - Graph traversal
- OpenMP 4.0 added task dependencies to influence scheduling

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `task` Construct

- Create a new task
  - Memorize data and code to be executed
  - Task constructs can be arbitrarily nested
  - By default tasks are *tied* to the thread that first executes them, not neccessarily the creator.
- Syntax (C/C++)

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

- Syntax (Fortran)

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


Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `task` clauses (1/2)

- Data scoping clauses:
  `shared(list)`, `private(list)`, `firstprivate(list)`,
  `default(private*|firstprivate*|shared|none)` * Fortran only
- Note: data scoping has a slightly different meaning than for threads
  - Private: variable is stored in the task's private memory

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `task` clauses (2/2)

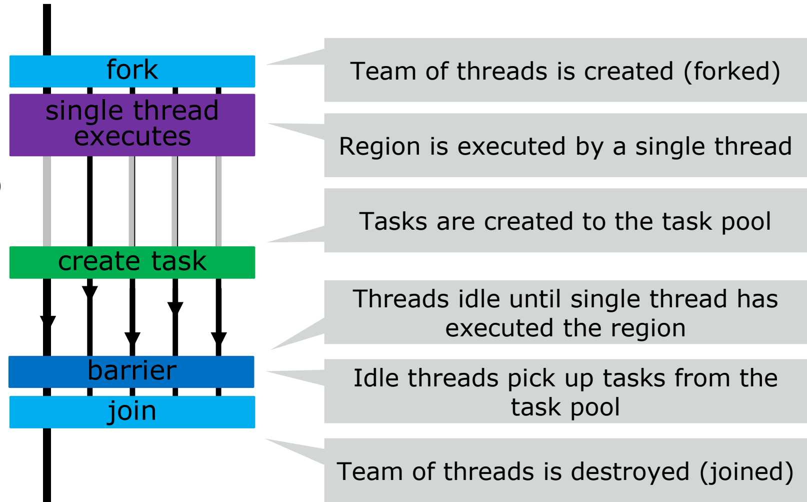
- Task scheduling clauses:
- `final(logical-expression)`
  The task is a *final task* if logical-expression evaluates to true. All tasks generated by final task are *included tasks*, i.e., will be evaluated immediately after their generation.
- `untied`
  The task is not tied to the thread that started its execution. Any thread in the team can resume the task region after a suspension.
- `mergeable`
  When the generated task is an *undeferred task* or an *included task*, the implementation may generate a *merged task*, i.e., a task for which the data environment is the same as its generating region.
- `depend(dependence-type:list)`

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `task`: Example (1/2)

```
!$omp parallel
  !$omp single
  e => l % head
  DO WHILE (ASSOCIATED(e))
    !$omp task &
    !$omp firstprivate(e)
      call process(e)
    !$omp end task
    e => e % next
  END DO
  !$omp end single
!$omp end parallel
```



Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP Task Scheduling: `taskwait`, `barrier`

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

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `task`: Example (2/2)

- Fibonacci numbers $F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$ with OpenMP tasks

```
INTEGER(KIND=int64) :: fn
INTEGER :: n
! Set n
!$omp parallel
  !$omp single
    fn = fibonacci(n)
  !$omp end single
!$omp single
…
RECURSIVE FUNCTION fibonacci(n) RESULT(fn)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER(KIND=int64) :: fn, fnm1, fnm2

  IF (n<2) THEN
    fn=n
    RETURN
  END IF
```

```
  ! Compute Fibonacci number by recursion
  !$omp task shared(fnm1)
    fnm1 = fibonacci(n-1)
  !$omp end task
  !$omp task shared(fnm2)
    fnm2 = fibonacci(n-2)
  !$omp end task

  !$omp taskwait
  fn = fnm1+fnm2
END FUNCTION fibonacci
```

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskgroup` Construct

- The `taskgroup` construct specifies a wait on the child tasks of the current task and their descendents at the end of construct
- Syntax (C/C++)

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

- Syntax (Fortran)

```
!$omp taskgroup
 block
!$omp end taskgroup
```

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskgroup`: Example

- Not waiting for all tasks currently in execution

```
!$omp parallel
  !$omp task
  CALL background_task()
  !$omp end task

  !$omp taskgroup
    !$omp task
    CALL compute_task()
   !$omp end task
  !$omp end taskgroup
  ! compute_task() and its siblings guaranteed to be finished
  !$omp taskwait
  ! background_task() guaranteed to be finished
!$omp end parallel
```

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `depend` clause

- Idea: programmer defines the dependencies between the tasks, the OpenMP runtime extracts the parallelism from the dependency graph
- The `depend` clause can enforce constraints on the order of execution by enabling dependencies between sibling tasks
  - By default the order of execution for OpenMP tasks is arbitrary

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `depend` clause

- Syntax:
  `depend(dependence-type:list)`
  where dependence-type is either `in`, `out` or `inout`, `list` describes one or more variables or non-zero length array sections.
- **Restriction:** any `list` items used in a `depend` clause of the same task or sibling tasks must indicate identical storage or disjoint storage, i.e., partially overlapping OpenMP array sections as dependencies are not allowed

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `depend` dependency types

- Task dependency types:
- `in`

  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, inout`

  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.

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP array sections

- An OpenMP array section designates the elements in an array
  - Array sections must be contiguous in memory
- Syntax (C/C++)

  `array[lower-bound:length]`

  alternatively

  `array[:length]`, `array[lower-bound:]`, `array[:]`
- Syntax (Fortran)

  `array(start:end)`

  or alternatively any **contiguous** Fortran array section

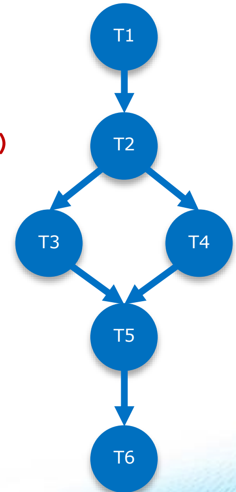
SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP depend: Example

```
INTEGER, TARGET :: a
INTEGER, POINTER :: b
b => a
!$omp task depend(in:a)
  ! T1
!$omp end task
!$omp task depend(out:b)
  ! T2
!$omp end task
!$omp task depend(in:a)
  ! T3
!$omp end task
```

```
!$omp task depend(in:b)
 ! T4
!$omp end task
!$omp task depend(inout:a)
 ! T5
!$omp end task
!$omp task depend(in:a)
 ! T6
!$omp end task
```



SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskloop` construct

- Motivation: OpenMP loop worksharing or using nested parallelism can lead to excessive fork/join overhead
  - Typical example: a call to a threaded numerical library such as BLAS from within an OpenMP parallel region

```
INTEGER :: N
REAL(kind=real64) :: A(N,N), B(N,N), C(N,N)
 !$omp parallel
 CALL compute_in_parallel_1(A,B,C)
 !$omp end parallel
 CALL DGEMM('N','N', n, n, n, &
                  real(1,real64), A, size(A,1), &
                  B, size(B,1), &
                  real(0,real64), C, size(C,1))
 !$omp parallel
 CALL compute_in_parallel_2(A,B,C)
 !$omp end parallel
```

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskloop` construct

- OpenMP `taskloop` construct
  - Split iterations of a loop into chunks, execute the chunks in parallel by using OpenMP tasks
  - Defines an implicit `taskgroup`, i.e., tasks generated by `taskloop` are *undeferred*
- Syntax (C/C++)

```
#pragma omp taskloop [simd] [clauses]
 for-loops
```

- Syntax (Fortran)

```
!$omp taskloop [simd] [clauses]
 do-loops
!$omp end taskloop [simd]
```

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskloop` clauses (1/2)

- Worksharing clauses
- **`grainsize`(`grain-size`)**
  Positive integer **`grain-size`** defines the number of iterations assigned to each created task to be at least **`grain-size`**, but less than two times the value of the **`grain-size`**.
- **`nogroup`**
  No implicit taskgroup region will be created.
- **`num_tasks`(`num-tasks`)**
  Positive integer **`num-tasks`** defines the number of created tasks as at least **`num-tasks`**. Each task must have at least one logical loop iteration.
- **`collapse`(`n`)**
  Positive integer **`n`** defines the number of nested loops associated with the taskloop construct.

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskloop` clauses (2/2)

- Clauses inherited from `parallel` and `task`
- Data scoping clauses:
  `shared(list), private(list), firstprivate(list), default(private*|firstprivate*|shared|none)`
- Task scheduling clauses
  `final(logical-expr), untied, mergeable`

* Fortran only

Copyright © 2018 Intel Corporation. All rights reserved.

OpenMP `taskloop`: Example

- Mixing different types of tasks

```
!$omp parallel
!$omp task
  call background_task() ! Can execute concurrently
  !$omp end task

  !$omp taskloop grainsize(500) nogroup
  DO i=1,N
     CALL computation_task(i) ! Can execute concurrently
  END DO
  !$omp end taskloop
!$omp end parallel
```

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Contents

- OpenMP* Threading Model
  - `parallel`, `do` and `task` constructs, `barrier`, data scoping, scheduling
- OpenMP* 4.0 Tasking
  - Task `depend`
  - OpenMP* 4.5 Tasking: `taskloop`
- OpenMP* performance aspects
  - Thread affinity
  - Non-uniform memory access

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

Thread or process pinning

- Pinning "assigns" a process or thread to a particular core for execution.
- If a thread/process is not pinned, the operating system is free to
  - initially start it on any core
  - freely move it around between cores
- Issues:
  - Might destroy memory locality (NUMA)
  - Sub-optimal placement might affect performance

Copyright © 2018 Intel Corporation. All rights reserved.

Thread affinity – processor binding

Success of processor binding strategies depend on the machine and the application!

- 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)

Copyright © 2018 Intel Corporation. All rights reserved.

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
- OpenMP 4.0 affinity policies
  - **spread** spread OpenMP threads evenly among the places
  - **close** pack OpenMP threads near master thread
  - **master** collocate OpenMP thread with master thread
- OpenMP 4.0 affinity control via
  - Environment variables **`OMP_PLACES`** and **`OMP_PROC_BIND`**
  - Clause **proc_bind** for parallel regions

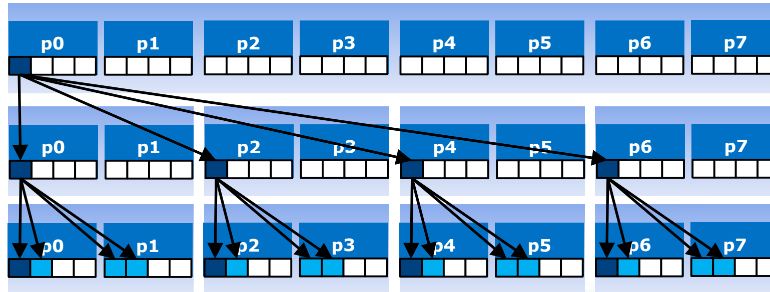
SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

Thread affinity: example

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

```
OMP_PLACES=cores(8)
!$omp parallel proc_bind(spread)
   !$omp parallel proc_bind(close)
```



SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

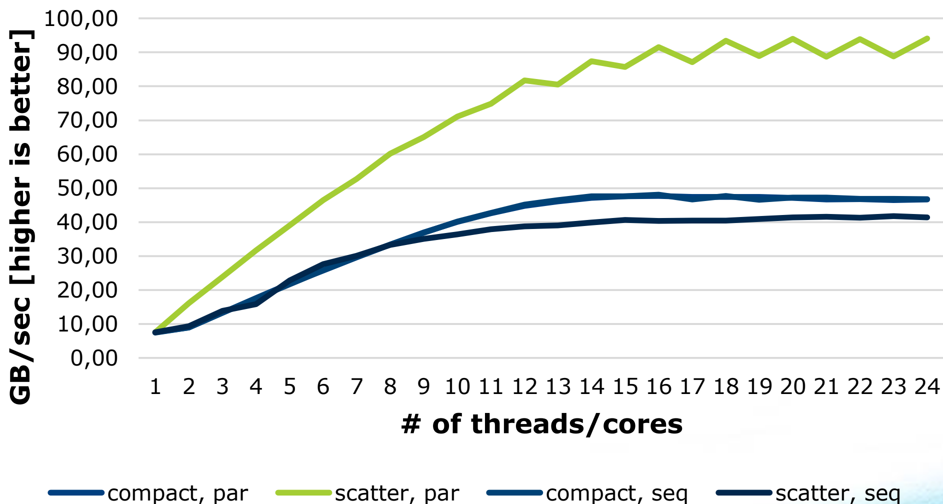
(Almost) all HPC systems are NUMA

- **N**on-**U**niform **M**emory **A**ccess
- (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 E5-2600v3 Series processor

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

NUMA - Does it matter?



SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

First touch policy

- Modern operating systems all use virtual memory
- The OS typically optimizes memory allocations
  - `malloc()` or **`ALLOCATE`** does not allocate the memory directly
  - Only the memory management "knows" about the memory allocation, but no memory pages are made available
  - At first memory access (***write***), the OS physically allocates the corresponding page (First touch policy)
- On NUMA systems this might lead to performance issues in threaded or multi-process applications

SOFTWARE AND SERVICES

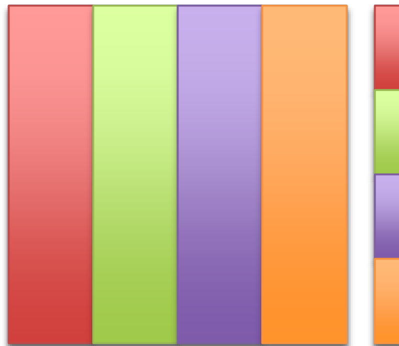
Copyright © 2018 Intel Corporation. All rights reserved.

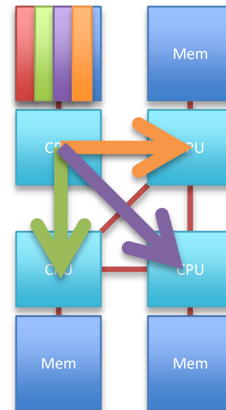
NUMA Optimization with OpenMP

```
! Initialize data

DO j=1,N
  DO i=1,M
  …
  END DO
END DO

! Perform work
!$omp parallel do private(j)
DO j=1,N
  DO i=1,M
  …
  END DO
END DO
!$omp end parallel do
```



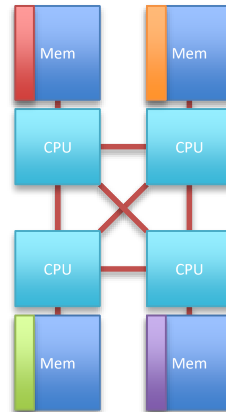


SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.

NUMA Optimization with OpenMP

```
! Initialize data
!$omp parallel do private(j)
DO j=1,N
  DO i=1,M
  …
  END DO
END DO
!$omp end parallel do
! Perform work
!$omp parallel do private(j)
DO j=1,N
  DO i=1,M
  …
  END DO
END DO
!$omp end parallel do
```



SOFTWARE AND SERVICES

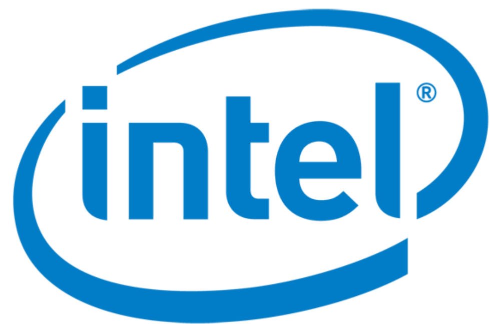
Copyright © 2018 Intel Corporation. All rights reserved.

Summary

- Use OpenMP tasks for irregular parallelism and for increased composability
- Task dependencies enable OpenMP runtime to extract the parallelism
- Task loops can be used to extract task parallelism from (nested) loops

SOFTWARE AND SERVICES

Copyright © 2018 Intel Corporation. All rights reserved.


intel