What's New In OpenMP

Edward Smyth
NAG

Technical Talk
ECMWF
2nd April 2019

Main OpenMP standards

- 1.0 October 1997 (Fortran), October 1998 (C/C++)
- 2.5 May 2005 (unified C/C++ and Fortran)
- 3.0 May 2008
- 4.0 July 2013
- 4.5 Nov 2015
- 5.0 Nov 2018
- Further major release every five years
  - With a minor update two years later

OpenMP Basics

OpenMP basics

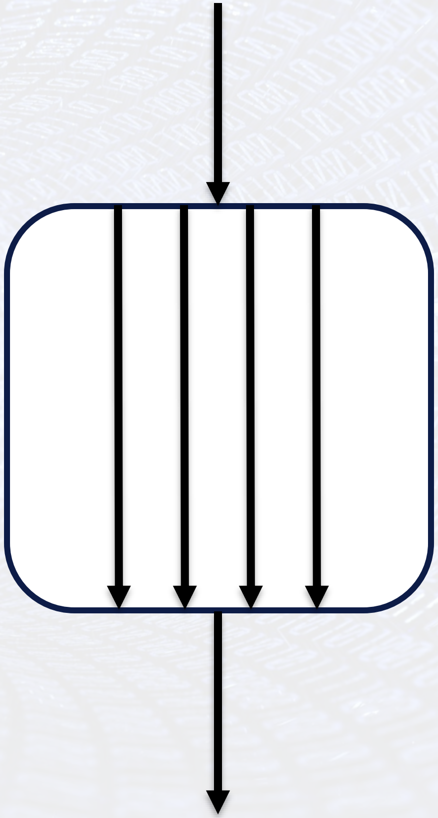
- Fortran and C/C++ support
- Compiler directives/pragmas + some library routines
  - ***!$omp parallel*** in Fortran, ***#pragma omp parallel*** in C/C++
- Internal Control Variables (ICV)
  - Default values (may be implementation-specific)
  - Can be set by environment variables or library calls
- Declarative and prescriptive semantics
  - Newer standards increase choices
- Increasing support for advanced hardware features
  - Vectorization, NUMA, heterogeneous architectures (e.g. GPUs)

"Fork-join" execution model

Serial execution

Multithreading in a ***parallel region.*** These threads form a "contention group"

Serial execution



Spawn threads

Destroy threads

Ways to implement parallelism in OpenMP

- 1. D.I.Y.

```
   ...
!$OMP PARALLEL PRIVATE (np,me)

   np = omp_get_num_threads()
   me = omp_get_thread_num()

   if (me==0) then
     ...

!$OMP END PARALLEL
    ...
```

- Other directives/pragmas can help
  - ***BARRIER, SINGLE, MASTER, ATOMIC***
- Lock routines supported
  - Simple and nested versions

Ways to express parallelism in OpenMP

2. Worksharing directives

```
!$OMP PARALLEL

!$OMP DO [clause...]
   ...
!$OMP END DO


!$OMP SECTIONS [clause...]
  !$OMP SECTION
     ...
  !$OMP SECTION
     ...
!$OMP END SECTIONS

!$OMP END PARALLEL
```

```
!$OMP PARALLEL

!$OMP WORKSHARE
   Y(:) = Y(:) + A*X(:)
!$OMP END WORKSHARE

!$OMP END PARALLEL
```

OpenMP *do/for* Scheduling

- Iterations can be divided among threads in a number of ways: Main two are ***static*** and ***dynamic***
  - ***Static***: Lowest runtime overhead (default choice)
  - ***Dynamic***: Higher overhead, but good for load balancing
- OpenMP 4.5 added ***monotonic*** and ***nonmonotonic*** qualifiers to ***dynamic***, latter is default in OpenMP 5.0
  - but may take time for compilers to implement
- If you use ***schedule(dynamic)*** and have a small amount of work per iteration, check out

  https://www.openmp.org/wp-content/uploads/SC18-BoothTalks-Cownie.pdf

Newer ways to express parallelism in OpenMP

- 3. Tasks (OpenMP 3.0, greatly enhanced in later)
- 4. "Doacross loops" (OpenMP 4.5)
  - Combines ideas from worksharing do/for loop and tasks
- 5. SIMD (OpenMP 4.0)
- 6. Device directives (OpenMP 4.0)
  - For heterogeneous systems

TASKS

omp tasks

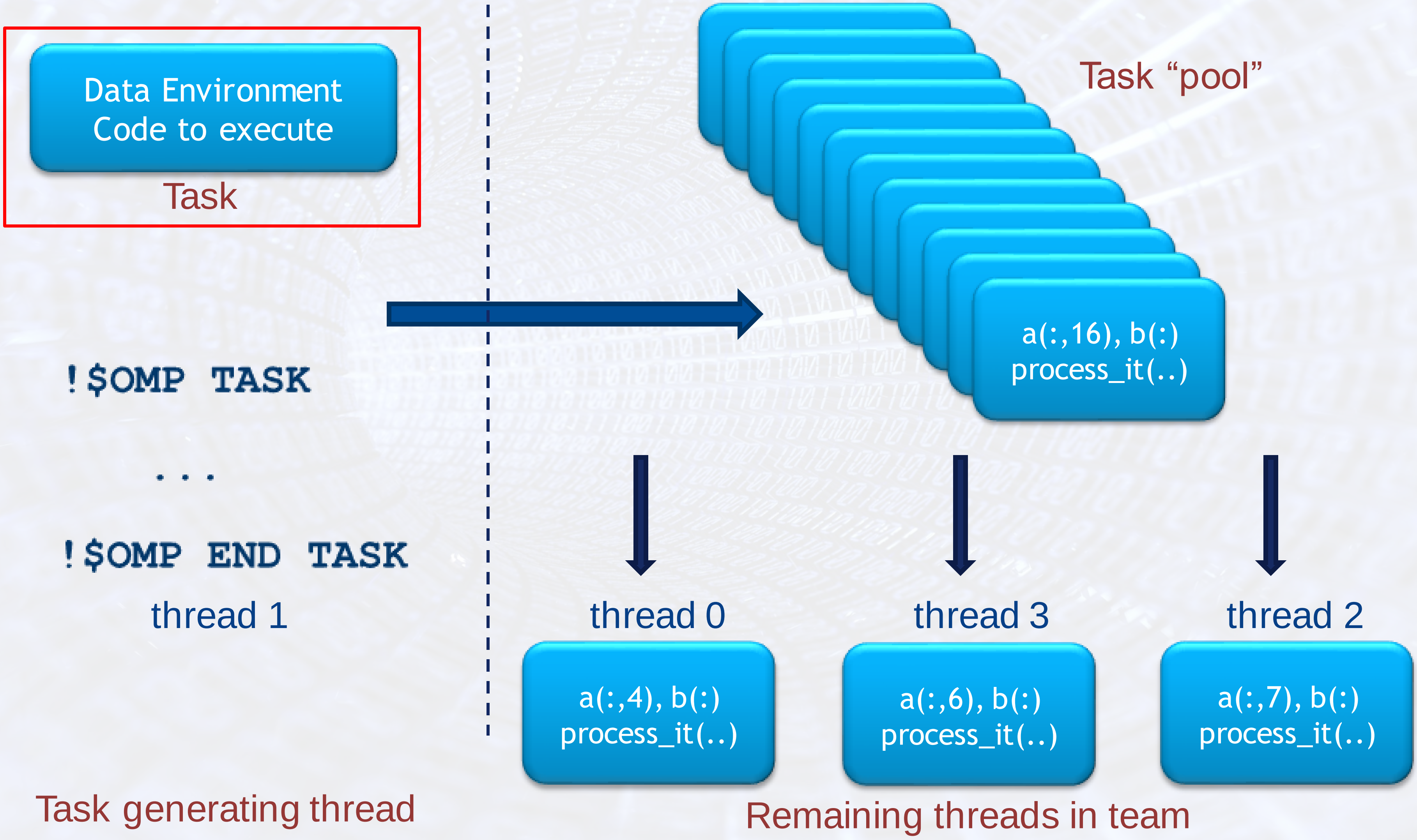
- A major addition to OpenMP at 3.0.
- Allow parallelization of irregular problems such as:
  - unbounded loops
  - producer/consumer schemes
  - recursive algorithms
  - linked lists
  - overlapping computation and I/O
- Tasks are work units that consist of some code to execute on some data, the *data environment.*

omp task example

```
Real     :: a(n,big_num), b(n)
Integer :: I

!$omp parallel default(none) shared(a,b) private(I)
!$omp single

  Do I = 1, big_num
     !$omp task
       Call process_it(a(:,I),b(:),n)
     !$omp end task
  End Do

!$omp end single
  ...
!$omp end parallel
```

Tasks will complete in the (implicit) barrier.

Task Example



Data Environment
Code to execute

Task

```
!$OMP TASK

  ...

!$OMP END TASK
```

thread 1

Task generating thread

Task “pool”

a(:,16), b(:)
process_it(..)

thread 0

a(:,4), b(:)
process_it(..)

thread 3

a(:,6), b(:)
process_it(..)

thread 2

a(:,7), b(:)
process_it(..)

Remaining threads in team

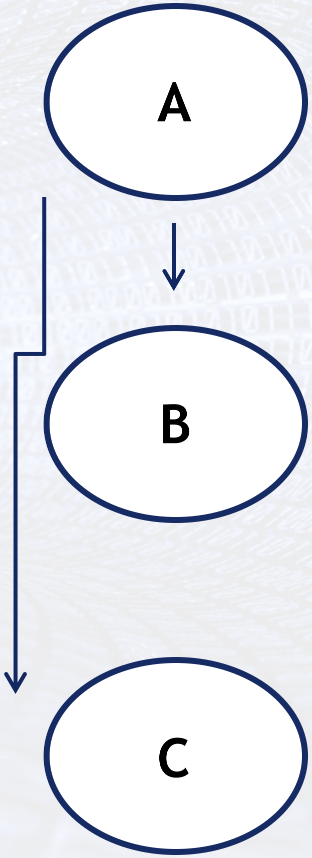
Task Dependencies (OpenMP 4.0)

- Very powerful new feature.
- The ***depend*** clause on the task directive enforces additional constraints on the scheduling of tasks.
- These constraints establish dependences, but only between *sibling* tasks.
- The clause consists of a dependence-type with one or more list items.
- Syntax:

```
depend( dependence-type : list )
```

Task Dependencies

- **Out** implies a write to x. This must take place before an **in**, which implies a read.
- So for these three tasks A must complete before tasks B and C. There is no other dependence.



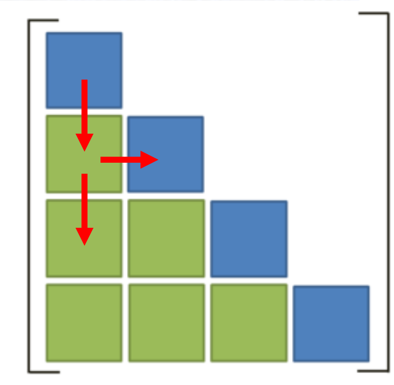
```
!$omp task … depend(out: x)
#pragma omp task … depend(out: x)
```

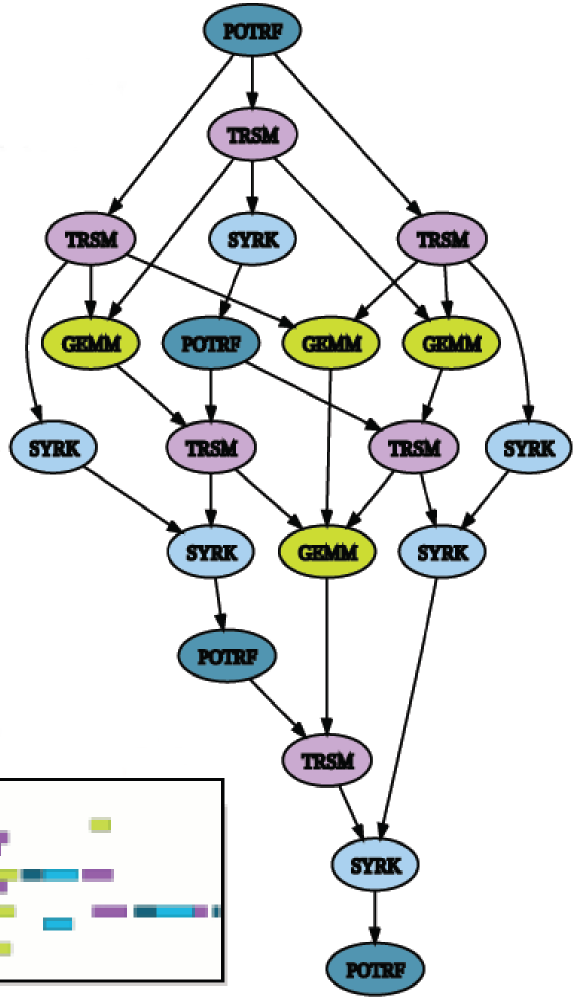
```
!$omp task … depend(in: x)
#pragma omp task … depend(in: x)
```

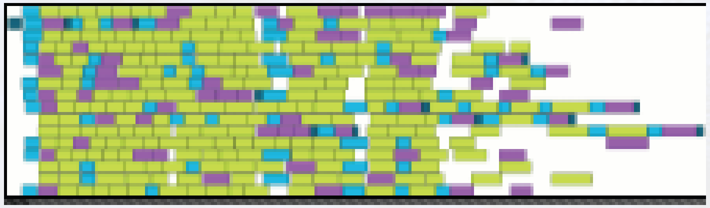
```
!$omp task … depend(in: x)
#pragma omp task … depend(in: x)
```

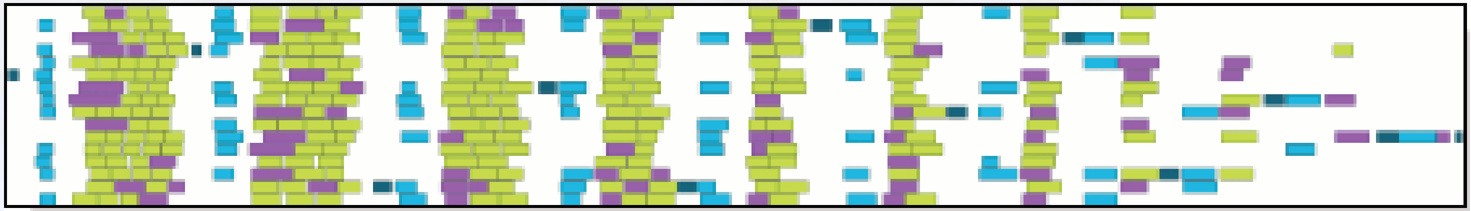
PLASMA Style Algorithms

- Task dependencies allow us to write tiled DAG based algorithms.
- 4 x 4 Cholesky









omp taskwait

- The ***taskwait*** construct specifies a wait on the completion of *child* tasks of the current task.
- The current task is suspended at this point until all child tasks generated before the ***taskwait*** complete execution.

```
#pragma omp single
{
  printf("A ");
  #pragma omp task
    {printf("race ");}
  #pragma omp task
    {printf("car");}
  #pragma omp taskwait
  printf("is fun to watch.\n");
}
```

omp taskgroup

- The ***taskgroup*** construct specifies a wait on the completion of *child* tasks of the current task and their descendent tasks.

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

- When a thread encounters a ***taskgroup*** construct it starts executing the region.

- The current task is suspended at the task scheduling point until all child tasks generated in the ***taskgroup*** region and all of their descendent tasks complete execution.

omp taskgroup

- Can prevent the synchronisation of all siblings that would occur with a ***taskwait***.

```
#pragma omp task
    {background_work();}
#pragma omp taskgroup
{
    #pragma omp task
        some_work();
    #pragma omp task
        some_more_work();
}
even_more_work();
```

background_work() and taskgroup can execute at the same time

Could contain tasks that will form part of the taskgroup

Group done, background_work() can continue beyond this point

Priority for explicit task (OpenMP 4.5)

- Priority hint can be provided on ***task*** (non-negative integer, 0 is default value):

```
!$omp task priority(priority-value)
   .. structured block ..
!$omp end task
```

- Using priorities does not guarantee execution order.
- Maximum value given is determined by ICV.
  - **min(priority-value, omp_get_max_task_priority())**
- Set ICV with **OMP_MAX_TASK_PRIORITY** env variable. Default 0.

omp taskloop (OpenMP 4.5)

- The ***taskloop*** construct specifies that the iterations of one or more loops will be executed in parallel using tasks.

```
!$omp taskloop [clause...]
  do-loops
!$omp end taskloop
```

- Usual task clauses (data sharing, final, untied, etc.) except ***depend***, plus:

grainsize	Specifies the minimum number of iterations per task
lastprivate	Usual meaning
num_tasks	Specifies the maximum number of tasks created
collapse	Specify the number of loops the directive applies to
nogroup	No implicit taskgroup is created

"Doacross loops"

Source and Sink

- The ***depend*** clause can now be used on ***do ordered***.
- Syntax:

```
depend( source )

depend( sink : vec )
```

- Example:

Must wait for (*i-1*)th iteration

```
!$omp do ordered(1)
   do i = 2, N
     A(i) = foo(i)

     !$omp ordered depend(sink: i-1)
        B(i) = bar(A(i), B(i-1))
     !$omp ordered depend(source)

     C(i) = baz(B(i))
   end do
!$omp end do
```

The *i*th iteration has “finished” with regard to dependencies

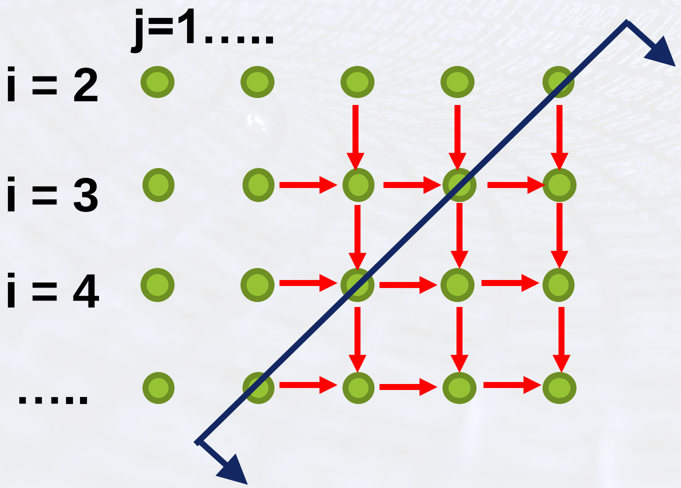
Source and Sink

```
!$omp do ordered(2)
  do j=2, N
   do i=2, M
     A(i,j) = foo(i, j)

    !$omp ordered depend(sink: j-1,i) depend(sink: j,i-1)
      B(i,j) = bar(A(i,j), B(i-1,j), B(i,j-1))
    !$omp ordered depend(source)
      C(i,j) = baz(B(i,j))
   end do
 end do
```

Do across applies to 2 loops

Must wait for two. Note *vector* of iterations

The (*i,j*)th iteration has "finished" with regard to dependencies



Wavefront parallelism

- Can also be implemented using ***omp tasks*** with dependencies
- Multifrontal example for Gauss-Seidel smoother given in webinar by Michael Klemm
  - https://techdecoded.intel.io/essentials/openmp-5-0-a-story-about-threads-and-tasks

SIMD

omp simd

- Informs compiler that vectorization of a loop is possible on a single thread

- Syntax:

```
!$OMP SIMD [clause]
  do-loops
!$OMP END SIMD
```

- Familiar clauses are ***private***, ***lastprivate***, ***reduction*** and ***collapse***. Also:
  - ***safelen*** Number of consecutive iterations without dependencies
  - ***aligned*** Specify how arrays are aligned
  - ***simdlen*** Hint to specify ideal number of iterations per vector
  - ***linear*** Variable are declared private and initialized to original value + the logical iteration number [X a step]

omp declare simd

- Tells compiler to create a vector function from a scalar one, so it can be called within an ***omp simd*** loop
- Syntax

```
!$OMP DECLARE SIMD [function/subroutine name][clause]
```

- Clauses aid compiler optimization, e.g.

simdlen	Number of iterations per vector (command, not hint)
uniform	Argument stays constant for all iterations
[not]inbranch	Will [never] be called within a conditional statement
aligned	Same as for omp simd
linear	Same as for omp simd

Combining directives

- The *SIMD* directive can be combined with loops and taskloops.
- Do SIMD syntax:

```
!$OMP DO SIMD [clause…]
  do-loops
!$OMP END DO SIMD
```

- Taskloop SIMD syntax:

```
!$OMP TASKLOOP SIMD [clause…]
  do-loops
!$OMP END TASKLOOP SIMD
```

- The clauses can be any accepted by either directive.

Device directives

Device directives

- To target heterogeneous systems
  - Initial release in OpenMP 4.0, major updates in 4.5
- No distinction made about device capabilities
  - Can have multiple devices per host
- Off-load model from host to accelerator device
  - Using ***omp target*** directive
  - By default, host waits for target region to complete
  - Device code will run on host if no accelerator present

Device directives

- Contention group: the set of threads (or *team*) descended from an initial thread
- Each ***omp target*** directive creates a new initial thread and thus a new contention group
- Threads in different contention groups cannot synchronize with each other
  - Except in a very limited way by using ***omp atomic***

Device directives

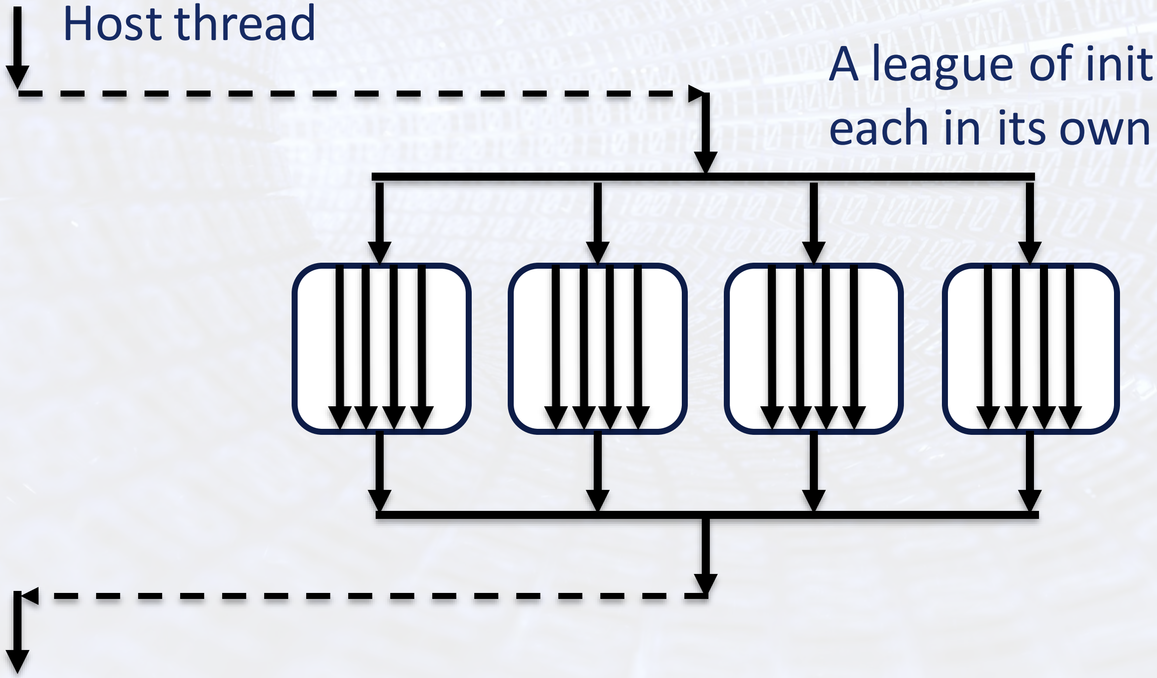
- ***omp parallel*** creates more threads within the same contention group
  - ***omp do*** to workshare across threads with a team

- ***omp teams*** starts a league of teams, each a separate initial thread in its own contention group
  - ***omp distribute*** to workshare across the league of teams

- Can be combined (optionally with ***omp simd***) to give up to three distinct types of parallelism
  - Which may map well to architecture of widely used accelerators, e.g. GPUs

Example

```
!$omp target teams distribute parallel do simd [clause,…]
   Do i = 1,n
     y(i) = y(i) + a*x(i)
   End do
!$omp end target teams distribute parallel do simd
```



Host thread

A league of initial threads each in its own team

Each initial thread become the master of a team of threads

Heterogeneous Memory Model

- Accelerator device and host may or may not share memory
  - ***Private***, ***firstprivate*** and automatic (stack) variables work as expected
    - Scalar variables default to ***firstprivate*** in OpenMP 4.5
  - Mapped variables (similar to ***shared*** variables on host)

```
!$omp target ... map(y(1:n)) map(to:x(1:n))
   Do i = 1,n
     y(i) = y(i) + a*x(i)
   End do
!$omp end target ...
```

- Other directives to help optimize performance, deal with pointers, synchronize with host

Device directives: support and performance

- Relatively immature in terms of compiler support
  - Feature implementation
  - Bugs
  - Performance
- Matt Martineau, Simon McIntosh-Smith et al
  - https://research-information.bristol.ac.uk/files/127657247/iwomp_arch.pdf
  - Note section on OpenMP 4.0 vs 4.5 differences
  - On NVIDIA P100, mini-apps were between ~1.2x and ~3.2x slower than CUDA implementations

AFFINITY

Affinity

- OpenMP now has an *affinity* model. That is, a mechanism to map threads to cores and ensure they are not moved once they are set.
- Each implicit task has a *place-partition-var* ICV which holds the available places
- Controlled via clause on ***omp parallel*** or environment variable:

```
!$omp parallel proc_bind(master|close|spread)

export OMP_PROC_BIND="spread, spread, close"
```

Affinity – Defining place with *OMP_PLACES*

- A list of places is specified with the **OMP_PLACES** environment variable.

- The *place-partition-var* ICV obtains its initial value from the **OMP_PLACES** value, and makes the list available to the execution environment.

- The value of **OMP_PLACES** can be one of two types of values: either an abstract name describing a set of places or an explicit list of places described by non-negative numbers.

OMP_PLACES

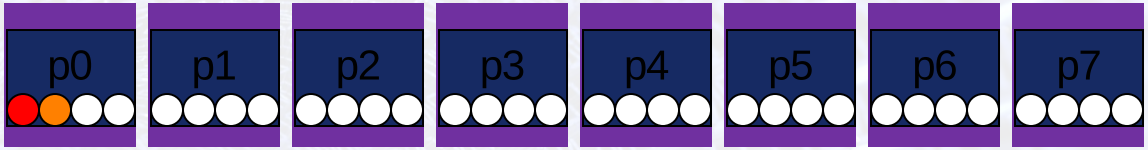
- Examples

```
export OMP_PLACES=sockets
export OMP_PLACES=cores
export OMP_PLACES=threads
export OMP_PLACES="threads(4)"

export OMP_PLACES=“{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
export OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"
export OMP_PLACES="{0:4}:4:4“
```

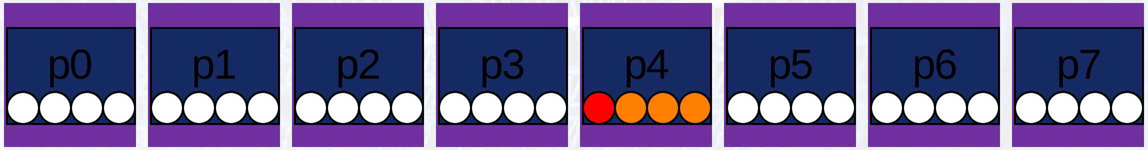
- where each of the last three definitions corresponds to the same 4 places including the smallest units of execution exposed by the execution environment numbered, in turn,
  0 to 3, 4 to 7, 8 to 11, and 12 to 15.

Affinity Examples - *Master*

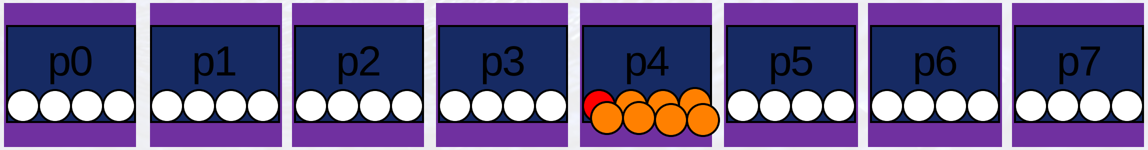
- 2 threads



- 4 threads



- 8 threads (over-subscription)

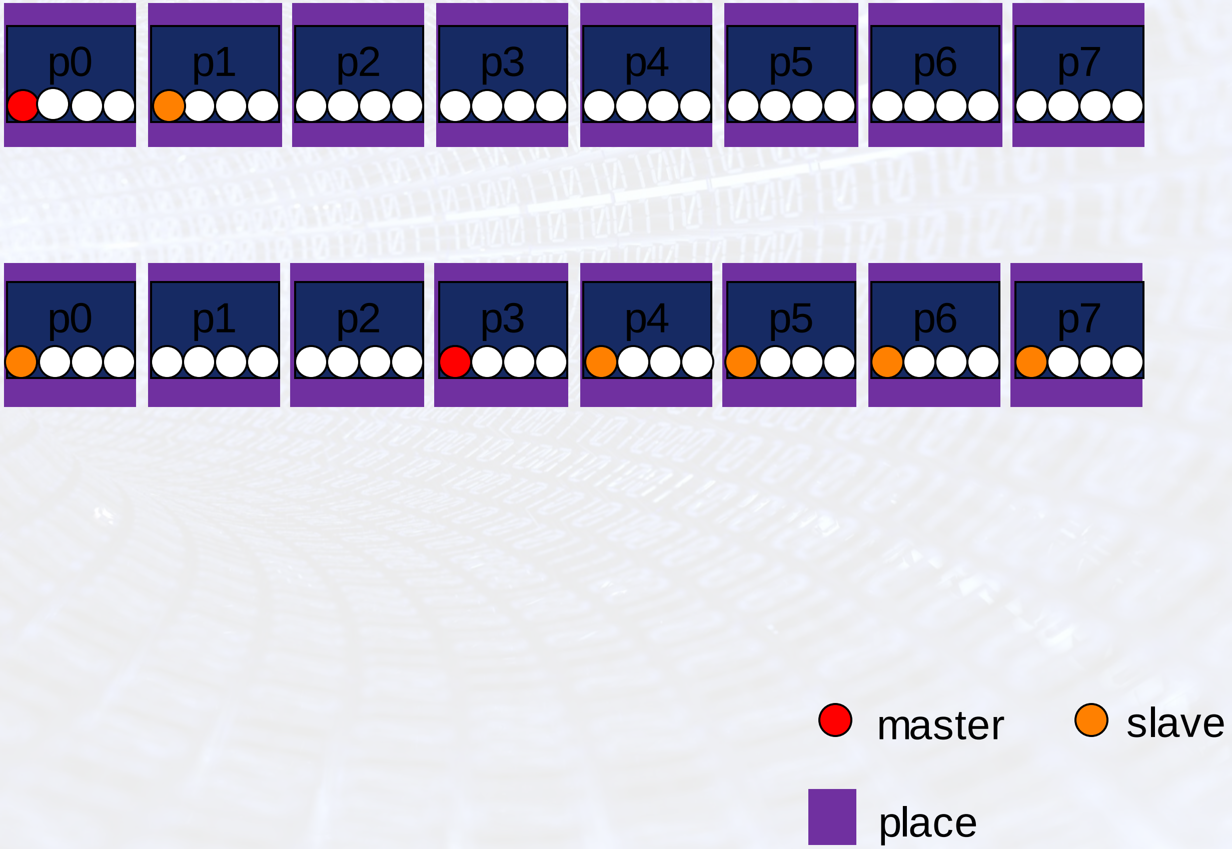


● master ● slave

■ place

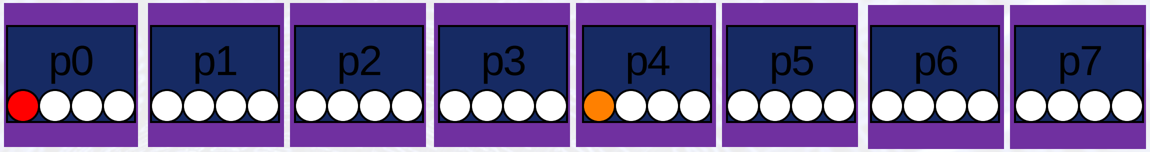
Affinity Examples - *Close*

- 2 threads
- 6 threads

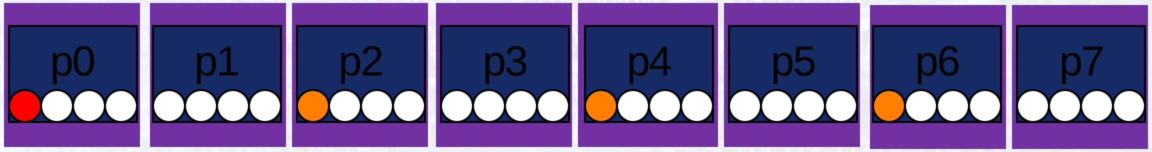


Affinity Examples - *Spread*

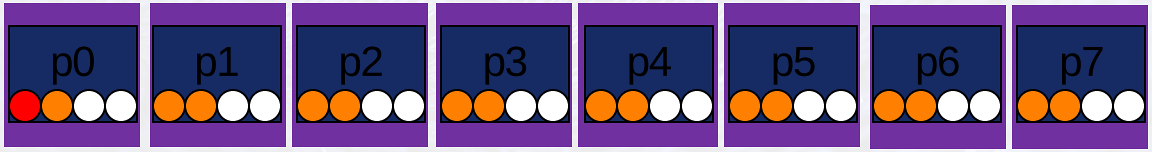
- 2 threads

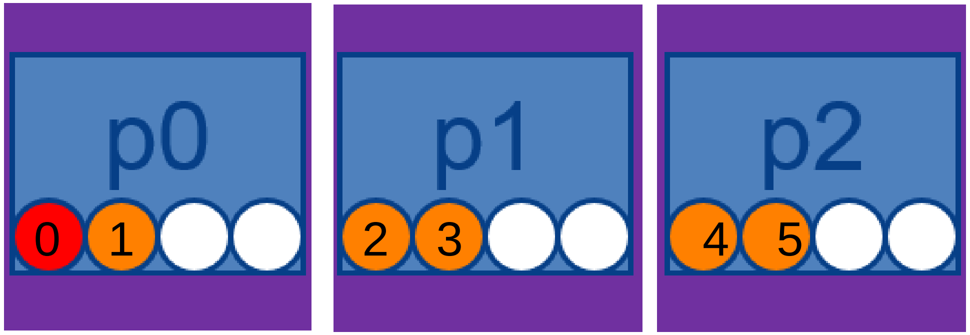


- 4 threads



- 16 threads





● master ● slave

▨ place

Memory management (OpenMP 5.0)

- Different memory spaces defined in the standard
  - *omp_default_mem_space*
  - *omp_large_cap_mem_space*
  - *omp_const_mem_space*
  - *omp_high_bw_mem_space*
  - *omp_low_lat_mem_space*
- Directives and library routines to manage memory allocation

Miscellaneous

Cancellation

- The ***cancel*** construct activates cancellation of the innermost enclosing region
  - That is, the innermost parallel, sections, do/for or taskgroup
- Helps with error handling, specifically propagating error flag from one thread to rest of the team
  - But doesn't interrupt other threads, all must check the cancellation points

OMP_DISPLAY_ENV (see also *OMP_DISPLAY_AFFINITY*)

- Set:

  export OMP_DISPLAY_ENV={TRUE,VERBOSE}

- Output

```
  OPENMP DISPLAY ENVIRONMENT BEGIN
    _OPENMP='201307'
    [host] OMP_SCHEDULE='GUIDED,4'
    [host] OMP_NUM_THREADS='4,3,2'
    [device] OMP_NUM_THREADS='2'
    [host,device] OMP_DYNAMIC='TRUE'
    [host] OMP_PLACES='{0:4},{4:4},{8:4},{12:4}'
    …
 OPENMP DISPLAY ENVIRONMENT END
```

Fortran language support

- The following features are *not* supported at 4.5 (but are in 5.0):
  - IEEE Arithmetic issues covered in Fortran 2003 Section 14
  - Parameterized derived types
  - The **PASS** attribute
  - Procedures bound to a type as operators
  - Overriding a type-bound procedure
  - Polymorphic entities
  - **SELECT TYPE** construct
  - Deferred bindings and abstract types
  - Controlling IEEE underflow
  - Another IEEE class value
- Improved Fortran 2008 support

Misc

- Improved C11 and C++11,14,17 support
- User-defined reductions (OpenMP 4.0)
  - Alternative to standard ones (+, -, max, min, etc)

Some other new OpenMP 5.0 features

- Loop ***collapse*** for imperfectly nested loops
- Task-to-data affinity hints
- Declarative vs prescriptive semantics:
  - ***metadirective*** and ***declare variant*** to allow choice of different directive code paths at runtime
  - ***omp loop***: Let compiler figure out best choice of ***do***, ***simd***, ***taskloop***, etc
- ***OMPT*** and ***OMPD*** interface for tools (debuggers, profilers, etc)
- ***OMP_NESTED*** is deprecated (replaced by ***OMP_MAX_ACTIVE_LEVELS***)

Compiler Support

▸ GCC

- From GCC 4.9.1, OpenMP 4.0 is fully supported.
- From GCC 6.1, OpenMP 4.5 is fully supported for C and C++

▸ Intel

- OpenMP 4.5 C/C++/Fortran supported in version 17.0, 18.0 and 19.0 compilers

▸ Cray

- OpenMP 4.5 supported in CCE 8.7 (April 2018)

▸ LLVM (Clang, Flang)

- Clang: OpenMP 4.5 (non-offloading) supported in Clang 3.9
- Flang: Much of OpenMP 4.5 supported

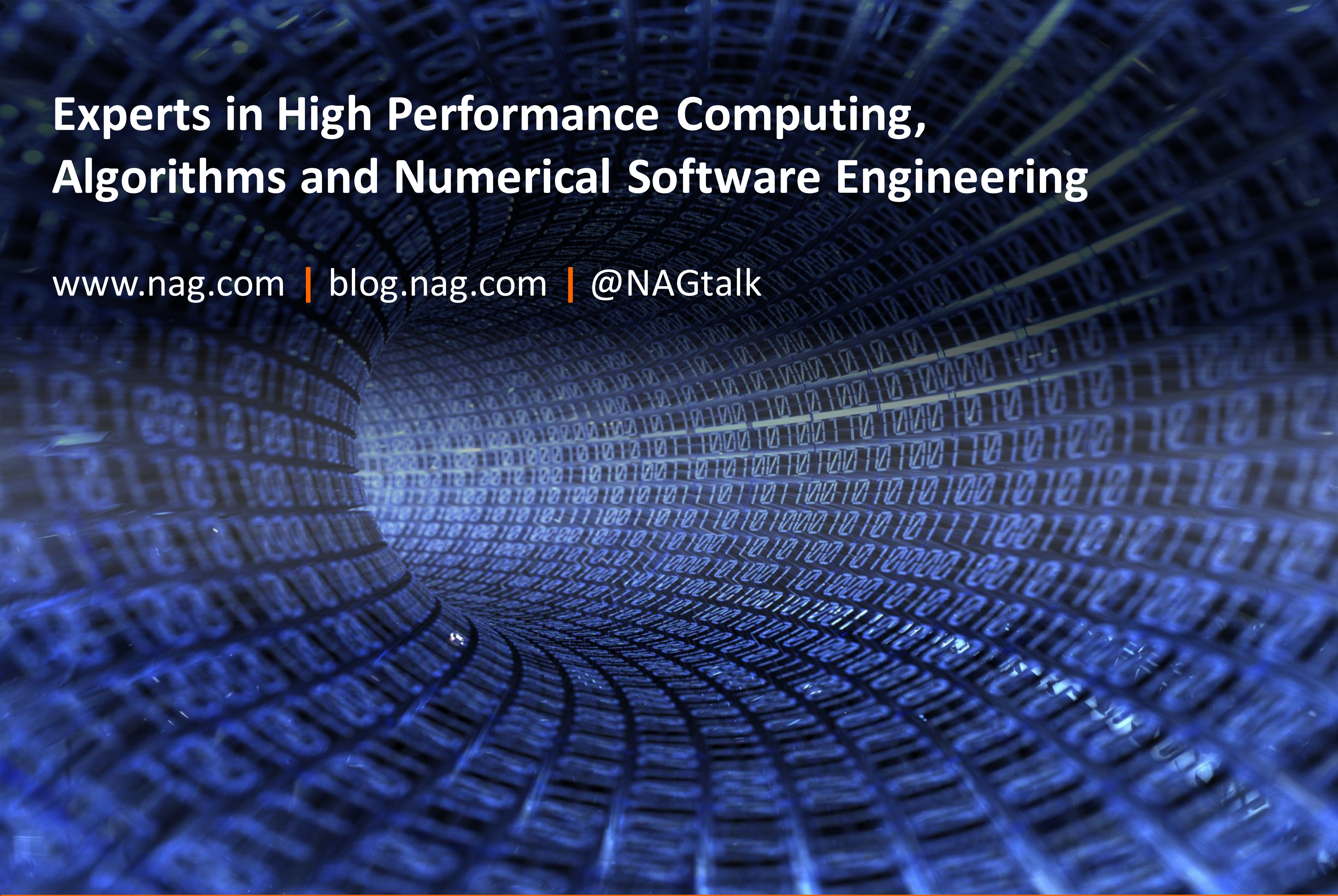
Compiler Support

- IBM
  - XLC/XLF 16.1.1 support OpenMP 4.5
- ARM
  - Full support for OpenMP 3.1, limited support for 4.0/4.5
- NAG
  - Fortran Compiler 6.2 supports OpenMP 3.1
- See here for more info:

  https://www.openmp.org/resources/openmp-compilers-tools/

Useful info

- ▶ OpenMP specifications and quick reference guides
  - https://www.openmp.org/specifications
  - https://www.openmp.org/resources/refguides
- ▶ "Using OpenMP – The Next Step" book (OpenMP 4.5)
- ▶ Michael Klemm webinar on OpenMP 5.0
  - https://techdecoded.intel.io/essentials/openmp-5-0-a-story-about-threads-and-tasks



Experts in High Performance Computing, Algorithms and Numerical Software Engineering

www.nag.com | blog.nag.com | @NAGtalk