

Advanced OpenMP

OpenMP 4.0/4.5



OpenMP 4.0

- Version 4.0 was released in July 2013
 - now available in most production compilers
- Version 4.5 was released in Nov 2015
 - corrections and a few new features
 - some full implementations

What's new in 4.0

- User defined reductions
- Construct cancellation
- Portable SIMD directives
- Extensions to tasking
- Thread affinity
- Accelerator offload support

User defined reductions

- As of 3.1 cannot do reductions on objects or structures.
- UDR extensions in 4.0 add support for this.
- Use **declare reduction** directive to define new reduction operators
- New operators can then be used in reduction clause.

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

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

Example

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

- Private copies created for a reduction are initialized to the identity that was specified for the operator and type
 - Default identity defined if identity clause not present
- Compiler uses combiner to combine private copies
- **omp_out** refers to private copy that holds combined values
- **omp_in** refers to the other private copy
- Can now use **merge** as a reduction operator.

Construct cancellation

- Clean way to signal early termination of an OpenMP construct.
 - one thread signals
 - other threads jump to the end of the construct

```
!$omp cancel construct [if (expr)]
```

where *construct* is **parallel**, **sections**, **do** or **taskgroup**
cancels the construct

```
!$omp cancellation point construct
```

checks for cancellation (also happens implicitly at cancel
directive, barriers etc.)

Example

```
!$omp parallel do private(eureka)
do i=1,n
    eureka = testing(i,...)
!$omp cancel parallel if(eureka)
end do
```

- First thread for which **eureka** is true will cancel the parallel region and exit.
- Other threads exit next time they hit the **cancel** directive

Portable SIMD directives

- Many compilers support SIMD directives to aid vectorisation of loops.
 - compiler can struggle to generate SIMD code without these
- OpenMP 4.0 provides a standardised set
- Use **simd** directive to indicate a loop should be SIMDized

#pragma omp simd [*clauses*]

- Executes iterations of following loop in SIMD chunks
- Loop is not divided across threads
- SIMD chunk is set of iterations executed concurrently by SIMD lanes

- Clauses control data environment, how loop is partitioned
- **safelen (length)** limits the number of iterations in a SIMD chunk.
- **linear** lists variables with a linear relationship to the iteration space
- **aligned** specifies byte alignments of a list of variables
- **private**, **lastprivate**, **reduction** and **collapse** have usual meanings.
- Also **declare simd** directive to generate SIMDised versions of functions.
- Can be combined with loop constructs (parallelise and SIMDise)

Extensions to tasking

- **taskgroup** directive provide allow task to wait for all descendant tasks to complete
- Compare **taskwait**, which only waits for children

```
#pragma omp taskgroup  
{  
    create_a_group_of_tasks(could_create_nested_tasks);  
} // all created tasks complete by here
```

Task dependencies

- **depend** clause on task construct

!\$omp task depend (*type* : *list*)

where *type* is **in**, **out** or **inout** and *list* is a list of variables.

- list may contain subarrays: OpenMP 4.0 includes a syntax for C/C++
- **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** or **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 **in**, **out** or **inout** clause.

Example

```
#pragma omp task depend (out:a)
  { ... }
#pragma omp task depend (out:b)
  { ... }
#pragma omp task depend (in:a,b)
  { ... }
```

- The first two tasks can execute in parallel
- The third task cannot start until both the first two are complete

Thread affinity

- Since many systems are now NUMA and SMT, placement of threads on the hardware can have a big effect on performance.
- Up until now, control of this in OpenMP is very limited.
- Some compilers have their own extensions.
- OpenMP 4.0 gives much more control

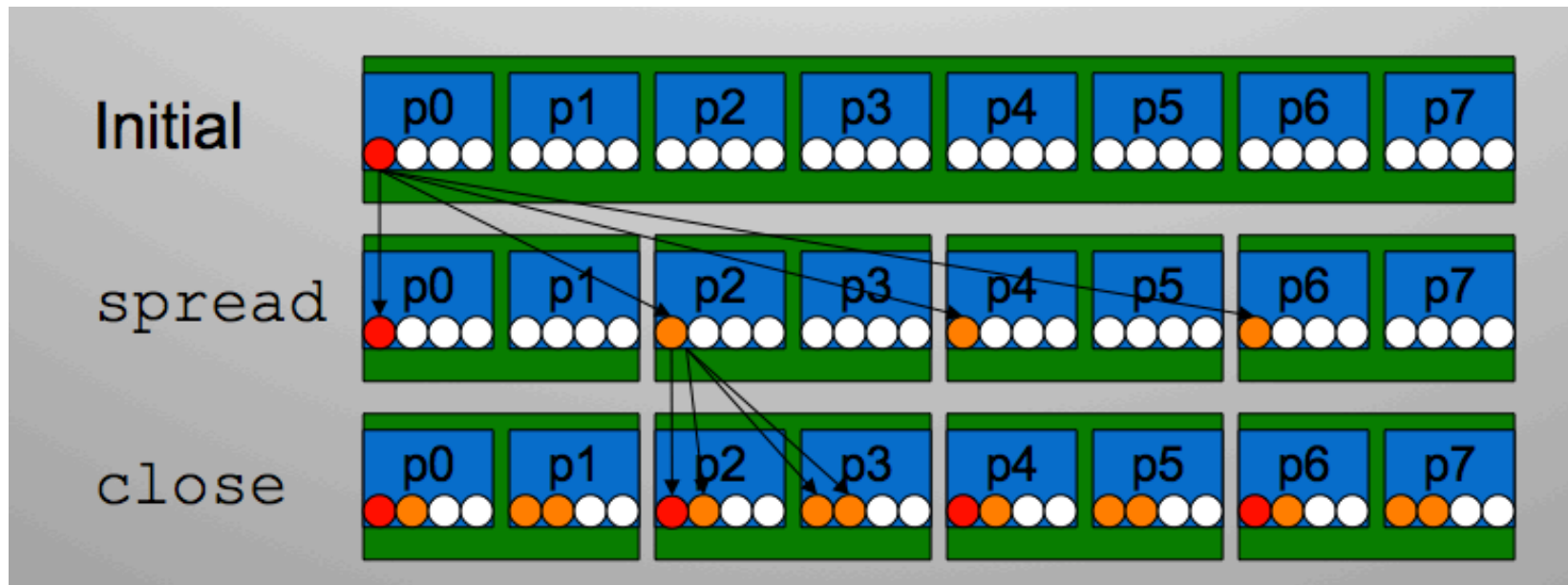
Affinity environment

- Increased choices for **OMP_PROC_BIND**
- Can still specify **true** or **false**
- Can now provide a list (possible item values: **master**, **close** or **spread**) to specify how to bind parallel regions at different nesting levels.
- Added **OMP_PLACES** environment variable
- Can specify abstract names including threads, cores and sockets
- Can specify an explicit ordered list of places
- Place numbering is implementation defined

Example

```
export OMP_PLACES=threads
```

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



Accelerator support

- Similar to, but not the same as, OpenACC directives.
- Support for more than just loops
- Less reliance on compiler to parallelise and map code to threads
- Not GPU specific
- Fully integrated into OpenMP

- Host-centric model with one host device and multiple target devices of the same type.
- *device*: a logical execution engine with local storage.
- *device data environment*: a data environment associated with a target data or target region.
- **target** constructs control how data and code is offloaded to a device.
- Data is mapped from a host data environment to a device data environment.

- Code inside target region is executed on the device.
- Executes sequentially by default.
- Can include other OpenMP directives to run in parallel
- Clauses to control data movement.

```
#pragma omp target map(to:B,C) , map(tofrom:sum)
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; i++){
    sum += B[i] + C[i];
}
```

- **target data** construct just moves data and does not execute code (c.f. **#pragma acc data** in OpenACC).
- **target update** construct updates data during a target data region.
- **declare target** compiles a version of function/subroutine that can be called on the device.
- Target regions are blocking: the encountering thread waits for them to complete.
 - Asynchronous behaviour can be achieved by using target regions inside tasks (with dependencies if required).

What about GPUs?

- Executing a target region on a GPU can only use one multiprocessor
 - synchronisation required for OpenMP not possible between multiprocessors
 - not much use!
- **teams** construct creates multiple master threads which can execute in parallel, spawn parallel regions, but not synchronise or communicate with each other.
- **distribute** construct spreads the iterations of a parallel loop across teams.
 - Only schedule option is static (with optional chunksize).

Example

```
#pragma omp target teams distribute parallel for\  
map(to:B,C) , map(tofrom:sum) reduction(+:sum)  
for (int i=0; i<N; i++){  
    sum += B[i] + C[i];  
}
```

- Distributes iterations across multiprocessors *and* across threads within each multiprocessor.