



# OpenMP 2.5 and beyond

**Mark Bull**

**OpenMP Architecture Review Board**

... also EPCC, University of Edinburgh

# OpenMP 2.5

- **Committee started work in March 2003**
  - ◆ fortnightly meetings
- **Very rough draft end of September 2003**
- **Moved to weekly meetings**
- **Version for public comment will be released for SC2004 (November)**

# How does it look?

- Fortran and C/C++ specifications have been merged.
  - ◆ as much common text as possible
  - ◆ language specific sections are identified graphically
  - ◆ defined common terms where possible (“variable”, “routine”, “structured block”)
  - ◆ merged text and resolved inconsistencies

# Significant changes

- Much expanded and reworked glossary
- Substantial reorganisation
- New material
  - ◆ Internal control variables
  - ◆ Memory model
  - ◆ Sharing attribute rules
- Resolution of some knotty problems
  - ◆ `flush (list)`
  - ◆ Cross-thread access to private variables
  - ◆ Persistence of `threadprivate`
  - ◆ Binding
- Improved examples

# Glossary

- clearer definition of concepts
- new improved terminology

# Regions and constructs

- The lexical extent of a directive is now called a *construct*.
- The dynamic extent of a directive is now called a *region*.
- A region is a dynamic instance of a construct
  - ◆ during the execution of a program a construct may give rise to many regions
- No such thing as a parallel region or critical section any more, but we can talk about `parallel` regions and `critical` regions
- Serial region -> *sequential part*

# More terminology...

- Binding is described with terms *binding thread set* and *binding region* (see later).
- Nesting is properly described by defining *nested region* and *nested construct*.
- Distinguish between *active* and *inactive* parallel regions.
- Definition of *variable* for both languages.
- Proper definition of *supporting nested parallelism*.

# Reorganisation

- **Made it clear and concise:**
  - ◆ rewritten confusing text and correct errors.
  - ◆ tried to resolve all inconsistencies between C/C++ and Fortran.
- **Tried to move all text relevant to a topic into one place.**
  - ◆ improve functionality as a reference
- **Removed some “tutorial” material, or moved it to examples**
- **Descriptions of constructs and library routines are more structured.**

# Internal control variables

- An implementation must behave as though there were internal variables which control the number of threads used to execute a `parallel` region and how to schedule loops.
- Default values, can be set by environment variables, can be set and read by library routines.
  - ◆ e.g. *nthreads-var* has a implementation defined default value, can be set by `OMP_NUM_THREADS` or `omp_set_num_threads()`, can be read by `omp_get_max_threads()`

- Other variables are *dyn-var*, *nest-var*, *run-sched-var* and *def-sched-var*
- Introduced decision diagrams to show how the values of the variables determine the number of threads used for every new `parallel` region, and the schedule to use for work-sharing loops.
- Hope that this will clarify how nested parallelism works.
- Allow the possibility for each thread to have its own value of these variables....

# Memory model

- The current specs don't specify this at all well...
- ...which means that nobody understands `flush`
- We have added a section to the spec which defines the relaxed memory model properly.

# Memory model

- A thread may have a local, *temporary view* of memory which is not always consistent with memory.
- A flush operation makes the temporary view consistent with memory:
  - ◆ A variable written since the last flush operation must be written back from the temporary view to memory
  - ◆ A variable not written since the last flush operation must be discarded from the temporary view, and subsequent read must come from memory

# Sharing attribute rules

- Material about whether variables are shared or private when not explicitly declared is scattered around and not very helpful.
  - ◆ e.g. Fortran loop control variables are private by default, but can they be made shared?
- Added new section making rules more explicit
  - Two cases:
    - ◆ variables referenced in a construct
    - ◆ variables referenced inside in a `parallel` region, but not in any construct.

## `flush(list)`

- `flush` with no list is no problem: it's a memory fence
  - ◆ All outstanding reads/writes must complete
  - ◆ No subsequent reads/writes can begin
- `flush(a)` means that
  - ◆ All outstanding reads/writes to `a` must complete
  - ◆ No subsequent reads/writes to `a` can begin
- This seems reasonable, but the 2.0 specs say nothing about ordering between `flush(a)` and reads/writes to other variables.

# Producer/consumer example

- Standard example for `flush(list)` from Chandra et al.

Producer:

```
data = ...
!$omp flush(data)
flag = 1
!$omp flush(flag)
```

Consumer:

```
do
!$omp flush(flag)
while (flag.eq.0)
!$omp flush(data)
... = data
```

There is nothing to prevent the write of flag occurring before the write of data!

# Rules for `flush(list)`

- Enforcing ordering between `flush(a)` and reads/writes to other variables makes it equivalent to `flush` with no list.

## New rules:

1. `flush` directives may be reordered with reads/writes to variables not in the list.
2. `flush` directives may be reordered with respect to each other provided they do not list any variables in common.

# Correct version

Producer:

```
data = ...  
!$omp flush(data,flag)  
flag = 1  
!$omp flush(flag)
```

Ensures data  
is written  
before flag



Consumer:

```
do  
!$omp flush(flag)  
while (flag.eq.0)  
!$omp flush(data,flag)  
... = data
```

Ensures data  
is read after  
flag



- Unfortunately, almost all examples/uses are incorrect, including the one in the 2.0 specs!

# Cross-thread access to private variables

- Long-standing unclear issue: is the following legal?

```
#pragma omp parallel shared(p) private(a,b)
{
  #pragma omp master
  {
    a = 23;
    p = &a;
  }
  #pragma omp barrier
  b = *p;
}
```

- Fortran and C/C++ 2.0 specs suggest that it's not legal, but aren't very clear.
- Original intentions of first language committees are also unclear.
- Making it legal has some attractions
  - ◆ Makes OpenMP more like Pthreads
  - ◆ Implementations have to support it to allow nested parallelism to work: a private variable can be shared in an inner parallel region
- However, it's not that simple:
  - ◆ doesn't work unless private variables are flushed
  - ◆ would need to specify the lifetime of private variables

# Decision time...

- After much deliberation and consideration of different possibilities, we voted to make it result in “unspecified behaviour”, except where the private variable belongs to an “ancestor” thread.
- Doesn't preclude allowing it at some time in the future....

# Persistence of threadprivate data

- Current specs say that if the number of threads is constant and dynamic parallelism is disabled, then threadprivate data values persist between `parallel` regions.
- But what happens in the case of nested parallelism?
- Does data persist between inner `parallel` regions (either within the same outer region, or between outer regions?)

# A conservative solution...

- Threadprivate data is only guaranteed to persist between two `parallel` regions if:
  1. Neither region is nested.
  2. The number of threads used is the same
  3. *dyn-var* is false for both regions and hasn't been changed
  4. *nthreads-var* is the same for both regions, and hasn't been changed

# Binding

- In 2.0 binding is a property of directives, but really it should be a property of regions
  - ◆ an orphaned construct may bind to several possible enclosing constructs
- Sometimes, we wish to use binding to talk about the threads which are affected.
  - ◆ e.g. `critical` affects all threads
- At other times, we wish to talk about the regions involved.
- Library routines give rise to regions also...

# Solution

- Every region has a *binding thread set* which is one of:
  - ◆ encountering thread
  - ◆ current team
  - ◆ all threads
- Regions whose binding thread set is the current team also have a *binding region* which is either
  - ◆ parallel region, or
  - ◆ loop region

# Examples

- Extended set of examples
- Match Fortran and C/C++ examples as closely as possible
- All correct examples are compilable

# OpenMP 3.0

- We have made some progress, but 2.5 has been harder than we thought...
- Some agreed proposals
  - ◆ Parallelisation of loop nests
  - ◆ Control of thread stack size
  - ◆ Control of idle thread behaviour
- Other topics in progress, but currently halted until 2.5 is done.
- Other issues have arisen from 2.5 discussions.

# Issues arising from 2.5 discussions

- **Cross-thread access to private variables.**
  - ◆ already discussed this..
- **Better support for controlling nested parallelism.**
- **Static schedule for work-sharing loops.**
- **Lock routines and flush.**
- **Fortran PURE procedures.**

# Support for nested parallelism

- The only way to specify the number of threads for an inner `parallel` region is with the `num_threads` clause.
- Would be better to allow `omp_set_num_threads` to have an effect in the current thread.
- Can do this by having per-thread copies of the internal control variables.
- Might need to add more getter/setter routines.

# Static schedule for work-sharing loops

- Is this safe?

```
#pragma omp for schedule(static) nowait
for (i=0; i<N; i++){
    a[i] = i;
}
#pragma omp for schedule(static)
for (i=0; i<N; i++){
    b[i] = a[i];
}
```

- Answer: NO!
- The current spec only guarantees *approximately equal* chunks, with one chunk per thread.
- Implementors want freedom within this to choose best chunks (e.g. for cache alignment)
- In 3.0 might want to constrain this more tightly, and allow freedom with new kinds of schedule.

# Lock routines and flush

- In 2.0 lock routines have no implied flush
  - ◆ Conscious decision by 2.0 committees: wanted to allow maximum optimisation.
- Unfortunately programmers seem completely oblivious to this!
  - ◆ Their codes work anyway....
  - ◆ Statement in 2.0 is rather weak...
  - ◆ Examples in 2.0 are wrong...
  - ◆ SPEC OMP benchmarks are wrong...
  - ◆ No mention in Chandra et al....

- **Shared variables accessed while a lock is held need to be flushed after the lock acquire and again before the lock release.**
  - ◆ **Expecting programmers to do this correctly using `flush(list)` is quite burdensome.**
  - ◆ **`flush(list)` is restricted to named variables: cannot flush array elements or pointees.**
  - ◆ **Also need to add rules about ordering of `flush` and the lock routines.**
- **Might be better to have implied flush on lock routines in 3.0**
  - ◆ **Put responsibility on compiler to figure out which variables actually need flushing.**

# Fortran PURE procedures

- Are OpenMP constructs or library routines allowed in PURE procedures?
- Is `omp_get_num_threads()` pure?
- In 3.0 will need to consider Fortran 2003 features in any case.

(We have tried to improve C++ support in 2.5, but more work may be required here, too)

# Summary

- OpenMP 2.5 is nearly ready.....
- We hope it is a significant improvement on the 2.0 specs, and a solid foundation for 3.0.
- We need your feedback during the public comment period.
  - ◆ please take a little time to give us your thoughts.