Shared Memory Programming with OpenMP

Lecture 8: Memory model, flush and atomics

Why do we need a memory model?

- On modern computers code is rarely executed in the same order as it was specified in the source code.
- Compilers, processors and memory systems reorder code to achieve maximum performance.
- Individual threads, when considered in isolation, exhibit as-ifserial semantics.
- Programmer's assumptions based on the memory model hold even in the face of code reordering performed by the compiler, the processors and the memory.



Reasoning about multithreaded execution is not that simple.

T1	Т2	
x=1;	int	r1=y;
y=1;	int	r2=x;

If there is no reordering and *T2* sees value of *y* on read to be 1 then the following read of *x* should also return the value 1.
If code in *T1* is reordered we can no longer make this assumption.

OpenMP Memory Model

- OpenMP supports a relaxed-consistency shared memory model.
 - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
 - These temporary views are made consistent only at certain points in the program.
 - The operation which enforces consistency is called the flush operation



- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
 - All previous read/writes by this thread have completed and are visible to other threads
 - No subsequent read/writes by this thread have occurred
 - A flush operation is analogous to a fence in other shared memory API's

Flush and synchronization

• A flush operation is implied by OpenMP synchronizations,

e.g.

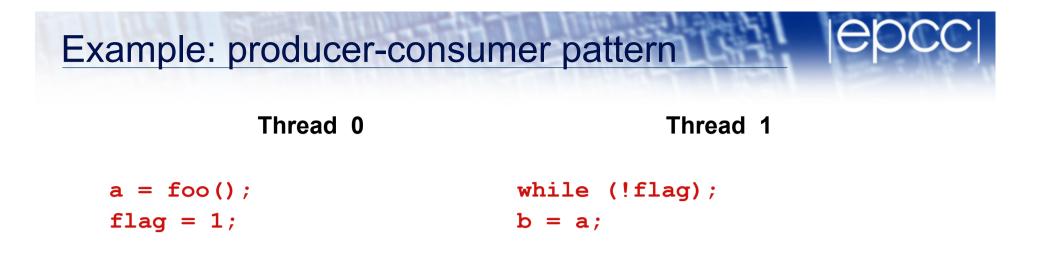
- at entry/exit of parallel regions
- at implicit and explicit barriers
- at entry/exit of critical regions
- whenever a lock is set or unset

• • • •

(but not at entry to worksharing regions or entry/exit of master regions)

• Note: using the **volatile** qualifier in C/C++ does *not* give sufficient guarantees about multithreaded execution.





- This is incorrect code
- The compiler and/or hardware may re-order the reads/writes to a and flag, or flag may be held in a register.
- OpenMP has a flush directive which specifies an explicit flush operation
 - can be used to make the above example work
 - !\$omp flush #pragma omp flush

Using flush

- In order for a write of a variable on one thread to be guaranteed visible and valid on a second thread, the following operations must occur in the following order:
 - 1. Thread A writes the variable
 - 2. Thread A executes a flush operation
 - 3. Thread B executes a flush operation
 - 4. Thread B reads the variable

Example: producer-consumer pattern



Thread 0

Thread 1

```
a = foo();
#pragma omp flush
flag = 1;
#pragma omp flush
```

```
#pragma omp flush
while (!flag) {
  #pragma omp flush
}
#pragma omp flush
b = a;
```

First flush ensures **flag** is written after **a**

Second flush ensures **flag** is written to memory

First and second flushes ensure **flag** is read from memory

Third flush ensures correct ordering of flushes

Using flush

- Using flush correctly is difficult and prone to subtle bugs
 - extremely hard to test whether code is correct
 - may execute correctly on one platform/compiler but not on another
 - bugs can be triggered by changing the optimisation level on the compiler
- Don't use it unless you are 100% confident you know what you are doing!
 - and even then.....

Other atomic forms

 Sometimes we may wish to enforce atomic behaviour for operations other than updates



Example: producer-consumer pattern Thread 1 Thread 0 #pragma omp flush a = foo();while (!myflag) { #pragma omp flush #pragma omp flush #pragma omp atomic write #pragma omp atomic read flag = 1;myflag = flag; #pragma omp flush #pragma omp flush b = a;

To be strictly correct we should use atomics to avoid the race condition on **flag**.