Lecture 15 — Memory Consistency

Patrick Lam & Jeff Zarnett
`patrick.lam@uwaterloo.ca`, `jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 18, 2019

OpenMP Memory Model

- All threads share a single store called memory.
  (may not actually represent RAM)
- Each thread has its own *temporary* view of memory.
- A thread's *temporary* view of memory
  is not required to be consistent with memory.

We'll talk more about memory models later.

Preventing Simultaneous Execution?

```
                    a = b = 0
/* thread 1 */                      /* thread 2 */

atomic(b = 1) // [1]                atomic(a = 1) // [3]
atomic(tmp = a) // [2]              atomic(tmp = b) // [4]
if (tmp == 0) then                  if (tmp == 0) then
    // protected section                // protected section
end if                              end if
```

Does this code actually prevent simultaneous execution?

Not seeing the problem...?

```
                    a = b = 0
/* thread 1 */                          /* thread 2 */

atomic(b = 1) // [1]                    atomic(a = 1) // [3]
atomic(tmp = a) // [2]                  atomic(tmp = b) // [4]
if (tmp == 0) then                      if (tmp == 0) then
    // protected section                    // protected section
end if                                  end if
```

Order				t1 tmp	t2 tmp
1	2	3	4	0	1
1	3	2	4	1	1
1	3	4	2	1	1
3	4	1	2	1	0
3	1	2	4	1	1
3	1	4	2	1	1

Looks like it (at least intuitively).

The Memory Model Contains Gotchas

```
                        a = b = 0
/* thread 1 */                              /* thread 2 */

atomic(b = 1) // [1]                        atomic(a = 1) // [3]
atomic(tmp = a) // [2]                      atomic(tmp = b) // [4]
if (tmp == 0) then                          if (tmp == 0) then
    // protected section                        // protected section
end if                                      end if
```

Sorry! With OpenMP's memory model, no guarantees:
the update from one thread may not be seen by the other.

Flush Ensures Consistent Views of Memory

`#pragma omp` **flush** *[(list)]*

Makes the thread's temporary view of memory consistent with main memory.

It enforces an order on memory operations of variables.

The variables in the list are called the flush-set.

If no variables given, compiler determines them for you.

Flush: Before is Before, After is After

Enforcing an order on the memory operations means:

- All read/write operations on the *flush-set* which happen before the **flush** complete before the flush executes.
- All read/write operations on the *flush-set* which happen after the **flush** complete after the flush executes.
- Flushes with overlapping *flush-sets* can not be reordered.

Flush Correctness

To show a consistent value for a variable between two threads, OpenMP must run statements in this order:

1. t_1 writes the value to v;
2. t_1 flushes v;
3. t_2 flushes v also;
4. t_2 reads the consistent value from v.

Take 2: Same Example, now improved with Flush

```
                              a = b = 0
/* thread 1 */                                  /* thread 2 */

atomic(b = 1)                                   atomic(a = 1)
flush(b)                                        flush(a)
flush(a)                                        flush(b)
atomic(tmp = a)                                 atomic(tmp = b)
if (tmp == 0) then                              if (tmp == 0) then
    // protected section                            // protected section
end if                                          end if
```

Will this now prevent simultaneous access?

No Luck Yet: Flush Fails

No.

- The compiler can reorder the `flush(b)` in thread 1 or `flush(a)` in thread 2.
- If `flush(b)` gets reordered to after the protected section, we will not get our intended operation.

Should you use flush?

Probably not, but now you know what it does.

Proper Use of Flush

```
                    a = b = 0
/* thread 1 */                        /* thread 2 */

atomic(b = 1)                         atomic(a = 1)
flush(a, b)                           flush(a, b)
atomic(tmp = a)                       atomic(tmp = b)
if (tmp == 0) then                    if (tmp == 0) then
    // protected section                  // protected section
end if                                end if
```

OpenMP Directives Where Flush Is Implied

- `omp barrier`
- at entry to, and exit from, **omp critical**;
- at exit from **omp parallel**;
- at exit from **omp for**;
- at exit from **omp sections**;
- at exit from **omp single**.

OpenMP Directives Where Flush Isn't Implied

- at entry to **for**;
- at entry to, or exit from, **master**;
- at entry to **sections**;
- at entry to **single**;
- at exit from **for**, **single** or **sections** with a **nowait**
    - **nowait** removes implicit flush along with the implicit barrier

This is not true for OpenMP versions before 2.5, so be careful.

Why Your Code is Slow

Want it to run faster? Avoid these pitfalls:

1. Unnecessary flush directives.
2. Using critical sections or locks instead of atomic.
3. Unnecessary concurrent-memory-writing protection:
   - No need to protect local thread variables.
   - No need to protect if only accessed in **single** or **master**.
4. Too much work in a critical section.
5. Too many entries into critical sections.

Example: Reducing Too Many Entries into Critical Sections

```
#pragma omp parallel for
for (i = 0; i < N; ++i) {
    #pragma omp critical
    {
        if (arr[i] > max) max = arr[i];
    }
}
```

would be better as:

```
#pragma omp parallel for
for (i = 0 ; i < N; ++i) {
    #pragma omp flush(max)
    if (arr[i] > max) {
        #pragma omp critical
        {
            if (arr[i] > max) max = arr[i];
        }
    }
}
```

OpenMP Wrap-up

Key points:

- How to use OpenMP **tasks** to parallelize unstructured problems.
- How to use **flush** correctly.

Memory Models

Sequential program: statements execute in order.
Your expectation for concurrency: sequential consistency.

> *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." — Leslie Lamport*

In brief:

1. for each thread: in-order execution;
2. interleave the threads' executions.

No one has it: too expensive; recall the worked example for **flush** last time.

Memory Models: Sequential Consistency

Another view of sequential consistency:

- each thread induces an *execution trace*.
- always: program has executed some prefix of each thread's trace.

The Blind Men and Elephant

But unfortunately, threads have their own view of the world.

Reordering

Compilers and processors may reorder non-interfering memory operations.

$$T1 : x = 1; r1 = y;$$

If two statements are independent:

- OK to execute them in either order.
- (equivalently: publish their results to other threads).

Reordering is a major compiler tactic to produce speedup.

Memory Consistency Models

Sequential consistency:

- No reordering of loads/stores.

Sequential consistency for datarace-free programs:

- If your program has no data races, then sequential consistency.

Relaxed consistency (only some types of reorderings):

- Loads can be reordered after loads/stores; and
- Stores can be reordered after loads/stores.

Weak consistency:

- Any reordering is possible.

Still, **reorderings** only allowed if they look safe in current context (i.e. independent; different memory addresses).

2011 Final Exam Question

```
                    x = y = 0

/* thread 1 */                      /* thread 2 */
x = 1;                              y = x;
r1 = y;                             r2 = x;
```

Assume architecture not sequentially consistent
(weak consistency).

Show me all possible (intermediate and final) memory values and how they arise.

2011 Final Exam Question: Solution

Must include every permutation of lines (since they can be in any order); then iterate over all the values.

Probably too long, but shows how memory reorderings complicate things.

The Compiler Reorders Memory Accesses

When it can prove safety, the **compiler** may reorder instructions (not just the hardware).

Example: want thread 1 to print value set in thread 2.

```
                              f = 0

/* thread 1 */                          /* thread 2 */
while (f == 0) /* spin */;              x = 42;
printf("%d", x);                        f = 1;
```

- If thread 2 reorders its instructions, will we get our intended result?

  No.

The Compiler Reorders Memory Accesses

When it can prove safety, the **compiler** may reorder instructions (not just the hardware).

Example: want thread 1 to print value set in thread 2.

```
                         f = 0

/* thread 1 */                          /* thread 2 */
while (f == 0) /* spin */;              x = 42;
printf("%d", x);                        f = 1;
```

- If thread 2 reorders its instructions, will we get our intended result?

  No.

KEEP OUT

Image Credit: MB298

Preventing Memory Reordering

A **memory fence** prevents memory operations from crossing the fence (also known as a **memory barrier**).

```
                              f = 0

/* thread 1 */                         /* thread 2 */
while (f == 0) /* spin */;             x = 42;
// memory fence                        // memory fence
printf("%d", x);                       f = 1;
```

- Now prevents reordering; get expected result.

Preventing Memory Reordering in Programs

Step 1: Don’t use volatile on C/C++ variables [1].

Syntax depends on the compiler.

- Microsoft Visual Studio C++ Compiler:

```
_ReadWriteBarrier()
```

- Intel Compiler:

```
    __memory_barrier()
```

- GNU Compiler:

```
__asm__ __volatile__ ("" ::: "memory");
```

The compiler also shouldn’t reorder across e.g. pthreads mutex calls.

[1] http://stackoverflow.com/questions/78172/using-c-pthreads-do-shared-variables-need-to-be-volatile.

Aside: gcc Inline Assembly

Just as an aside, here's gcc's inline assembly format

```
__asm__ ( assembler template
       : output operands                  /* optional */
       : input operands                   /* optional */
       : list of clobbered registers      /* optional */
       );
```

Last slide used **__volatile__** with __asm__. This isn't the same as the normal C volatile. It means:

- The compiler may not reorder this assembly code and put it somewhere else in the program.

Memory Fences: Preventing HW Memory Reordering

Memory barrier: no access after the barrier becomes visible to the system (i.e. takes effect) until after all accesses before the barrier become visible.

Note: these are all `x86` `asm` instructions.

`mfence`:

- All loads and stores before the fence finish before any more loads or stores execute.

`sfence`:

- All stores before the fence finish before any more stores execute.

`lfence`:

- All loads before the fence finish before any more loads execute.

Preventing Hardware Memory Reordering (Option 2)

Some compilers also support preventing hardware reordering:

- Microsoft Visual Studio C++ Compiler:

```
MemoryBarrier();
```

- Solaris Studio (Oracle) Compiler:

```
__machine_r_barrier();
__machine_w_barrier();
__machine_rw_barrier();
```

- GNU Compiler:

```
__sync_synchronize();
```

Memory Barriers and OpenMP

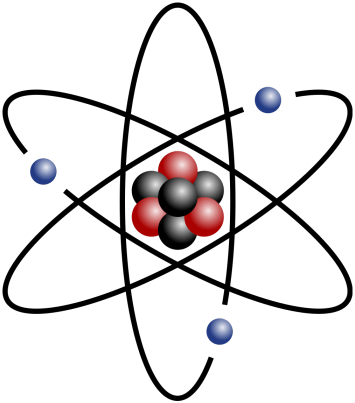
Fortunately, an OpenMP **flush** (or, better yet, mutexes) also preserve the order of variable accesses.

Stops reordering from both the compiler and hardware.

For GNU, flush is implemented as `__sync_synchronize();`

Note: proper use of memory fences makes `volatile` not very useful (again, `volatile` is not meant to help with threading, and will have a different behaviour for threading on different compilers/hardware).

Atomic Operations



`https://commons.wikimedia.org/w/index.php?curid=1675352`

Atomic Operations

We saw the **atomic** directive in OpenMP, plus C++11 atomics.

Most OpenMP atomic expressions map to atomic hardware instructions.

Other atomic instructions exist.

Compare and Swap

Also called **compare and exchange** (cmpxchg instruction).

```
int compare_and_swap (int* reg, int oldval, int newval) {
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

- Afterwards, you can check if it returned oldval.
- If it did, you know you changed it.

Implementing a Spinlock

Use compare-and-swap to implement spinlock:

```
void spinlock_init(int* lock) { *lock = 0; }

void spinlock_lock(int* lock) {
    while(compare_and_swap(lock, 0, 1) != 0) {}
    __asm__ ("mfence");
}

void spinlock_unlock(int* lock) {
    __asm__ ("mfence");
    *lock = 0;
}
```

You’ll see **cmpxchg** quite frequently in the Linux kernel code.

ABA Problem

Sometimes you'll read a location twice.

If the value is the same, nothing has changed, right?

ABA Problem

Sometimes you'll read a location twice.

If the value is the same, nothing has changed, right?

No. This is an **ABA problem**.

You can combat this by "tagging": modify value with nonce upon each write.

Can keep value separately from nonce; double compare and swap atomically swaps both value and nonce.

Waterloo...

The ABA problem is not any sort of acronym nor a reference to this:
`https://www.youtube.com/watch?v=Sj_9CiNkkn4`

ABA

It’s a value that is A, then changed to B, then changed back to A.

The ABA problem is a big mess for the designer of lock-free Compare-And-Swap routines.

1. P_1 reads A_i from location L_i.
2. P_k interrupts P_1; P_k stores the value B into L_i.
3. P_j stores the value A_i into L_i.
4. P_1 resumes; it executes a false positive CAS.

False Positive

It’s a “false positive” because P_1’s compare-and-swap operation succeeds even though the value at L_i has been modified in the meantime.

If this doesn’t seem like a bad thing, consider this.

If you have a data structure that will be accessed by multiple threads, you might be controlling access to it by the compare-and-swap routine.

What should happen is the algorithm should keep trying until the data structure in question has not been modified by any other thread in the meantime.

But with a false positive we get the impression that things didn’t change, even though they really did.

Napoleon was defeated...

You can combat this by “tagging”: modify value with nonce upon each write.

You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce.

Another example of this: Java `ConcurrentModificationException` is detected by checking the modification count of a collection.