

Memory Models and OpenMP

Hans-J. Boehm



Disclaimers:

- Much of this work was done by others or jointly. I'm relying particularly on:
 - Basic approach: Sarita Adve, Mark Hill, Ada 83 ...
 - JSR 133: Also Jeremy Manson, Bill Pugh, Doug Lea
 - C++0x: Lawrence Crowl, Clark Nelson, Paul McKenney, Herb Sutter, ...
 - Improved hardware models: Bratin Saha, Peter Sewell's group, ...
- But some of it is still controversial.
 - This reflects my personal views.
- I'm not an OpenMP expert (though I'm learning).
- My experience is not primarily with numerical code.

The problem

- Shared memory parallel programs are built on shared variables visible to multiple threads of control.
- But what do they mean?
 - Are concurrent accesses allowed?
 - What is a concurrent access?
 - When do updates become visible to other threads?
 - Can an update be partially visible?
- There was much confusion. ~2006:
 - Standard compiler optimizations “broke” C code.
 - Posix committee members disagreed about basic rules.
 - Unclear the rules were implementable on e.g. X86.
 - ...

Outline

- Emerging consensus:
 - Interleaving semantics (Sequential Consistency)
 - But only for data-race-free programs
- Brief discussion of consequences
 - Software requirements
 - Hardware requirements
- How OpenMP fits in:
 - Largely sequentially consistent for DRF.
 - But some remaining differences.
 - Current flush-based formulation is problematic.

Naive threads programming model (Sequential Consistency)

- Threads behave as though their memory accesses were simply interleaved. (Sequential consistency)

Thread 1

`x = 1;`

`z = 3;`

Thread 2

`y = 2;`

– might be executed as

`x = 1; y = 2; z = 3;`

Locks/barriers restrict interleavings

Thread 1

```
lock(l);  
r1 = x;  
x = r1+1;  
unlock(l);
```

Thread 2

```
lock(l);  
r2 = x;  
x = r2+1;  
unlock(l);
```

– can only be executed as

```
lock(l); r1 = x; x = r1+1; unlock(l); lock(l);  
r2 = x; x = r2+1; unlock(l);
```

or

```
lock(l); r2 = x; x = r2+1; unlock(l); lock(l);  
r1 = x; x = r1+1; unlock(l);
```

since second lock(l) must follow first unlock(l)

But this doesn't quite work ...

- Limits reordering and other hardware/compiler transformations
 - “Dekker’s” example (everything initially zero) should allow $r1 = r2 = 0$:

Thread 1

```
x = 1;  
r1 = y;
```

Thread 2

```
y = 1;  
r2 = x;
```

- Sensitive to memory access granularity:

Thread 1

```
x = 300;
```

Thread 2

```
x = 100;
```

- may result in $x = 356$ with sequentially consistent byte accesses.

Real threads programming model

- An interleaving exhibits a *data race* if two consecutive steps
 - access the same scalar variable*
 - at least one access is a store
 - are performed by different threads
- **Sequential consistency only for data-race-free programs!**
 - Avoid anything else.
- Data races are prevented by
 - locks (or atomic sections) to restrict interleaving
 - declaring synchronization variables (stay tuned ...)

} conflict

Data Races

- Are defined in terms of sequentially consistent executions.
- If x and y are initially zero, this does *not* have a data race:

Thread 1
if (x)
 y = 1;

Thread 2
if (y)
 x = 1;

Synchronization variables

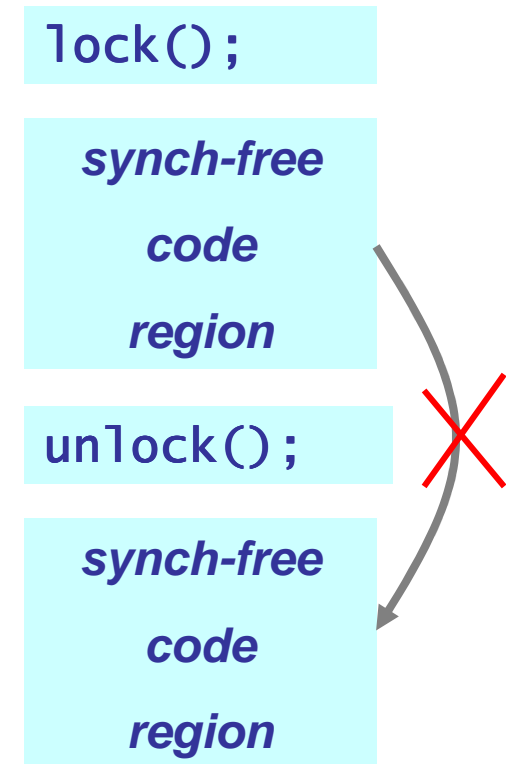
- Java: `volatile`, `java.util.concurrent.atomic`.
- C++0x: `atomic<int>`
- C1x: `atomic_int` → `_Atomic(int)`
- OpenMP 4.0 proposal:
 - `#pragma omp atomic seq_cst`.
- Guarantee indivisibility of operations.
- “Don’t count” in determining whether there is a data race:
 - Programs with “races” on synchronization variables are still sequentially consistent.
 - Though there may be “escapes” (Java, C++0x, not discussed here).
- Dekker’s algorithm “just works” with synchronization variables.

SC for DRF programming model advantages over SC

- Supports important hardware & compiler optimizations.
- DRF restriction → Independence from memory access granularity.
 - Hardware independence.
 - Synchronization-free library calls are atomic.
 - Really a different and better programming model than SC.

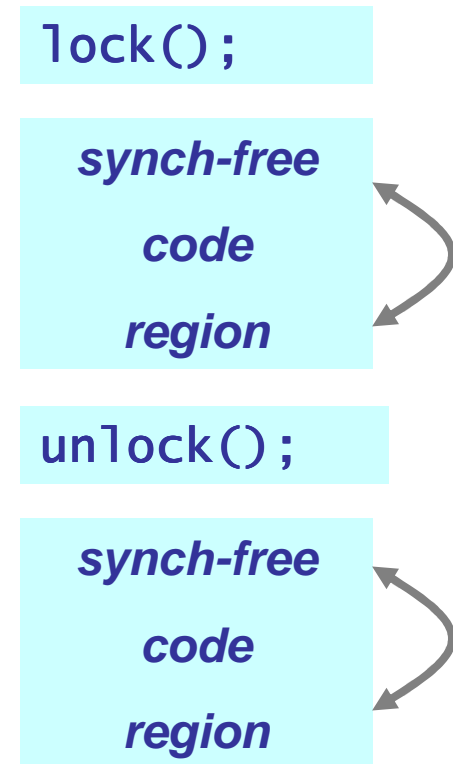
Basic SC for DRF implementation model (1)

- Sync operations sequentially consistent.
- Very restricted reordering of memory operations around synchronization operations:
 - Compiler either understands these, or treats them as opaque, potentially updating any location.
 - Synchronization operations include instructions to limit or prevent hardware reordering.
 - Usually “memory fences” (unfortunately?)



SC for DRF implementation model (2)

- Code may be reordered between synchronization operations.
 - Another thread can only tell if it accesses the same data between reordered operations.
 - Such an access would be a data race.
- If data races are *disallowed* (e.g. Posix, Ada, C++0x, OpenMP 3.0, *not* Java), *compiler may assume that variables don't change asynchronously.*



Possible effect of “no asynchronous changes” compiler assumption:

```
unsigned x;  
  
If (x < 3) {  
    ... // async x change  
    switch(x) {  
        case 0: ...  
        case 1: ...  
        case 2: ...  
    }  
}
```

- Assume switch statement compiled as branch table.
- May assume x is in range.
- Asynchronous change to x causes wild branch.
 - Not just wrong value.
- Rare, but possible in current compilers?



Some variants

C++ draft (C++0x) C draft (C1x)	SC for DRF*, Data races are errors
Java	SC for DRF**, Complex race semantics
Ada83+, Posix threads	SC for drf (sort of)
OpenMP, Fortran 2008	SC for drf (except atomics, sort of)
.Net	Getting there, we hope ☺

* Except explicitly specified memory ordering. ** Except some j.u.c.atomic.

An important note

- SC for DRF is a major improvement, but *not* the whole answer.
- There are serious remaining problems for
 - Debugging.
 - Programs that need to support “sand-boxed” code, e.g. in Java.
- We really want
 - sequential consistency for data-race-free programs.
 - at worst fail-stop behavior for data races.
- But that’s a hard research problem, and a different talk.

Outline

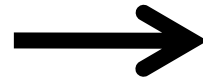
- Emerging consensus:
 - Interleaving semantics (Sequential Consistency)
 - But only for data-race-free programs
- **Brief discussion of consequences**
 - **Software requirements**
 - Hardware requirements
- How OpenMP fits in:
 - Largely sequentially consistent for DRF.
 - But some remaining differences.
 - Current flush-based formulation is problematic.

Compilers must not introduce data races

- Single thread compilers currently may add data races: (PLDI 05)

```
struct {char a; char b} x;
```

```
x.a = 'z';
```



```
tmp = x;  
tmp.a = 'z';  
x = tmp;
```

– `x.a = 1` in parallel with `x.b = 1` may lose `x.b` update.

- Still broken in gcc in bit-field-related cases.

A more subtle way to introduce data races

```
int count;    // global, possibly shared
...
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++count;
```



```
int count;    // global, possibly shared
...
reg = count;
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++reg;
count = reg;  // may spuriously assign to count
```

Synchronization primitives need careful definition

- More on this later ...

Outline

- Emerging consensus:
 - Interleaving semantics (Sequential Consistency)
 - But only for data-race-free programs
- Brief discussion of consequences
 - Software requirements
 - **Hardware requirements**
- How OpenMP fits in:
 - Largely sequentially consistent for DRF.
 - But some remaining differences.
 - Current flush-based formulation is problematic.

Byte store instructions

- `x.c = 'a';` may not visibly read and rewrite adjacent fields.
- Byte stores must be implemented with
 - Byte store instruction, or
 - Atomic read-modify-write.
 - Typically expensive on multiprocessors.
 - Often cheaply implementable on uniprocessors.

Sequential consistency must be enforceable

- Programs using only synchronization variables must be sequentially consistent.
- Usual solution: Add fences.
- Unclear that this is sufficient:
 - Wasn't possible on X86 until the re-revision of the spec last year.
 - Took months of discussions with PowerPC architects to conclude it's (barely, sort of) possible there.
 - Itanium requires other mechanisms.
- The core issue is “write atomicity”:

Can fences enforce SC?

Unclear that hardware fences can ensure sequential consistency. “IRIW” example:

x, y initially zero. Fences between every instruction pair!

<i>Thread 1:</i> <code>x = 1;</code>	<i>Thread 2:</i> <code>r1 = x; (1)</code> <code>fence;</code> <code>r2 = y; (0)</code>	<i>Thread 3:</i> <code>y = 1;</code>	<i>Thread 4:</i> <code>r3 = y; (1)</code> <code>fence;</code> <code>r4 = x; (0)</code>
---	---	---	---

x set first!

y set first!

Fully fenced, not sequentially consistent. Does hardware allow it?

Outline

- Emerging consensus:
 - Interleaving semantics (Sequential Consistency)
 - But only for data-race-free programs
- Brief discussion of consequences
 - Software requirements
 - Hardware requirements
- How OpenMP fits in:
 - Largely sequentially consistent for DRF.
 - But some remaining differences.
 - Current flush-based formulation is problematic.

So what about OpenMP?

- OpenMP 2.5 memory model was different.
 - Some reference examples used data races.
 - Explicit “volatile” support with unusual semantics.
 - Implicit flush on wrong(?) side of volatile.
 - Wouldn't work for common use cases.
- OpenMP 3.0 clearly states that
 - data races are disallowed.
 - volatile is only in base language.
- OpenMP 3.0 is still unclear in places
 - Some examples include data races, etc.
 - → fix in 3.1. *We make favorable assumptions* 😊

OpenMP 3.0/3.1 memory model*

- Currently states that it is a weak memory model.
 - But we'll reinterpret it.
- States it's based on release consistency.
 - Questionable. (Stay tuned.)
- Basically promises sequential consistency for many data-race-free programs.

* Intent is the same for 3.0 and 3.1; 3.1 will be clearer.

Why is OpenMP basically SC for DRF?

- Implied flush operations for synchronization operations exc. atomic
 - Synchronization operations sequentially consistent.
 - Prevents reordering across synchronization operations.
- Satisfies SC for DRF implementation rules.
- Execution equivalent to interleaving in which data ops occur just before next sync. op in thread.
 - If not, there is a data race.

But not fully SC for DRF

- 1) Current spec allows adjacent field overwrites.
- 2) Operations using `pragma omp atomic` do not preserve sequential consistency.
- 3) The 3.0 spec makes some subtle and dubious promises that are often not kept.
 - If we want consistency between spec and implementations, the spec should be restructured. (Maybe weasel-worded for 3.1?)
 - If we then also want SC for DRF, another subtle spec change is needed.
- Many users should care about (2), fewer about (1) and (3).

(1) Adjacent field overwrites

- Current spec permits implementations to allow:

```
char a[2];  
a[1] = 1; // May read and re-write a[0]
```

- This essentially makes it impossible to write fully portable parallel programs.
- Most implementations do this at most for bit-fields.
- We'll ignore the problem here.



(2) The problem with OpenMP atomics

Thread 1:

```
x = 42;  
#pragma omp atomic write  
x_ready = true;
```

Thread 2:

```
do {  
    #pragma omp atomic read  
    tmp = x_ready;  
} while (!tmp);  
assert(x == 42);
```

- Assertion may fail
- Memory accesses may be visibly reordered
 - In spite of absence of data races
- (Would work with release consistency.)
- May add `seq_cst` clause in 4.0

Explicit flushes as a workaround

- Easy to forget.
- More expensive on some architectures (e.g. x86, Itanium) than `seq_cst` atomics
- Consider

```
do {  
    #pragma omp atomic read  
    tmp = x_ready;  
} while (!tmp);  
#pragma omp flush  
assert(x == 42);
```

- `#pragma omp flush` requires expensive fence instruction.
 - To guard against prior atomic store.
- Sequentially consistent load does not.
- Add `seq_cst` clause in 4.0?

(3) With flush-based semantics, current atomics make locks expensive

(variant of Dekker's algorithm core)

x, y initially zero

Thread 1:

```
#pragma omp critical a
{
    #pragma omp atomic
    x = 1;
}
#pragma omp atomic
r1 = y;
```

Thread 2:

```
#pragma omp critical b
{
    #pragma omp atomic
    y = 1;
}
#pragma omp atomic
r2 = x;
```

- Implied flush ensures ordering !?
 - $r1 = r2 = 0$ outcome must be precluded!
- Common implementations of lock release provide no such guarantee.
- Broken on many platforms?

Similar issue for `omp_test_lock()` ?

Thread 1:

```
x = 42;  
omp_set_lock(&l);
```

Thread 2:

```
while (omp_test_lock(&l))  
    omp_unset_lock(&l);  
assert(x == 42);
```

- Assertion may fail if thread 1 statements are reordered.
 - Movement into critical section forbidden, but common?
- Preventing this is expensive.
- Can be avoided by allowing spurious `omp_test_lock()` failures. Status quo?

Future directions

- Explicitly provide SC for DRF guarantee?
 - Clearer description.
 - Consistent with other languages.
 - Far easier to reason about.
- Add `seq_cst` atomics for 4.0?
- Use a more conventional happens-before-based memory model for 4.0 to handle legacy atomics?
- Explicit `#pragma omp flush` basically matters only for current atomic operations.
 - What's its future role?

Questions?

6/16/2010

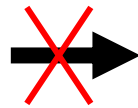


Backup slides

Introducing Races: Register Promotion 1

[g is global]

```
for(...) {  
    if(mt) lock();  
    use/update g;  
    if(mt) unlock();  
}
```



```
r = g;  
for(...) {  
    if(mt) {  
        g = r; lock(); r = g;  
    }  
    use/update r instead of g;  
    if(mt) {  
        g = r; unlock(); r = g;  
    }  
}  
g = r;
```

Why reordering between sync ops is OK

```
first = 1;  
second = 2;
```

Thread 1: ↓

```
second = 2;  
  
first = 1;
```

- How can reordering be detected?
- Only if intermediate state is observed.
 - By a racing thread!

Thread 2:

```
tmp1 = second; // 2  
tmp2 = first; // 0
```

Replace fences completely?

Synchronization variables on X86

- atomic store: *~1 cycle* *dozens of cycles*
 - store (*mov*); *mfence*;
- atomic load: *~1 cycle*
 - load (*mov*)
- Store implicitly ensures that prior memory operations become visible before store.
- Load implicitly ensures that subsequent memory operations become visible later.
- Sole reason for *mfence*: Order atomic store followed by *atomic* load.

Fence enforces all kinds of additional, unobservable orderings

- `s` is a synchronization variable:

```
x = 1;
```

```
s = 2; // includes fence
```

```
r1 = y;
```

- Prevents reordering of `x = 1` and `r1 = y`;
 - final load delayed until assignment to `a` visible.
- But this ordering is invisible to non-racing threads
 - ...and expensive to enforce?
- *We need a tiny fraction of `mfence` functionality.*

Dynamic Race Detection

- Need to guarantee one of:
 - Program is data-race free and provides SC execution (done),
 - Program contains a data race and raises an exception, or
 - Program exhibits simple semantics anyway, e.g.
 - Sequentially consistent
 - Synchronization-free regions are atomic
- This is significantly cheaper than fully accurate data-race detection.
 - Track byte-level R/W information
 - Mostly in cache
 - As opposed to epoch number + thread id per byte

For more information:

- Boehm, “Threads Basics”, HPL TR 2009-259.
- Boehm, Adve, “Foundations of the C++ Concurrency Memory Model”, PLDI 08.
- Sevcik and Aspinall, “On Validity of Program Transformations in the Java Memory Model”, ECOOP 08.
- Owens, Sarkar, Sewell, “A better X86 Memory Model: x86-TSO”, TPHOLs 2009.
- S. V. Adve, Boehm, “Memory Models: A Case for Rethinking Parallel Languages and Hardware”, to appear, CACM.
- Lucia, Strauss, Ceze, Qadeer, Boehm, “Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions“, to appear, ISCA 10.

Trylock restricts `lock()` reordering:

- Some really awful code:

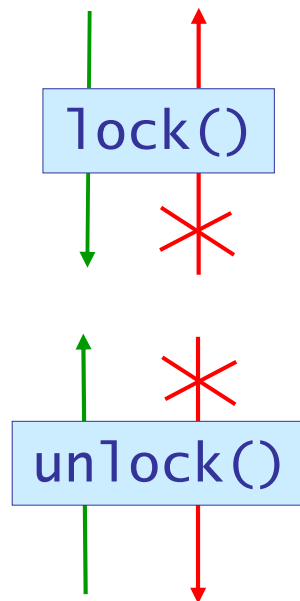
<i>Thread 1:</i>	<i>Thread 2:</i>	<i>Don't try this at home!!</i>
<pre>x = 42; lock();</pre>	<pre>while (trylock() == SUCCESS) unlock(); assert (x == 42);</pre>	

- Can't move `x = 42` into critical section!
- Disclaimer: Example requires tweaking to be pthreads-compliant.

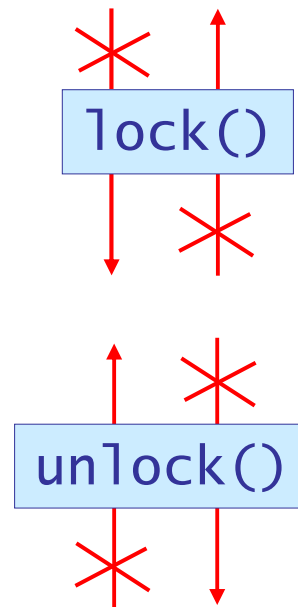
Trylock: Critical section reordering?

- Reordering of memory operations with respect to critical sections:

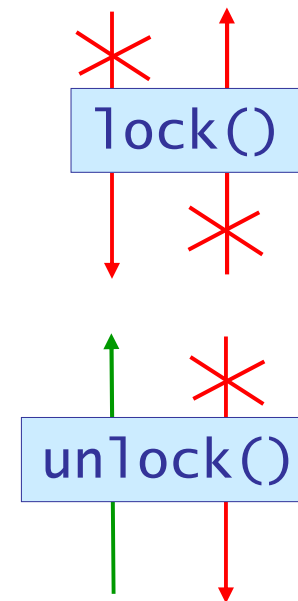
Expected (& Java):



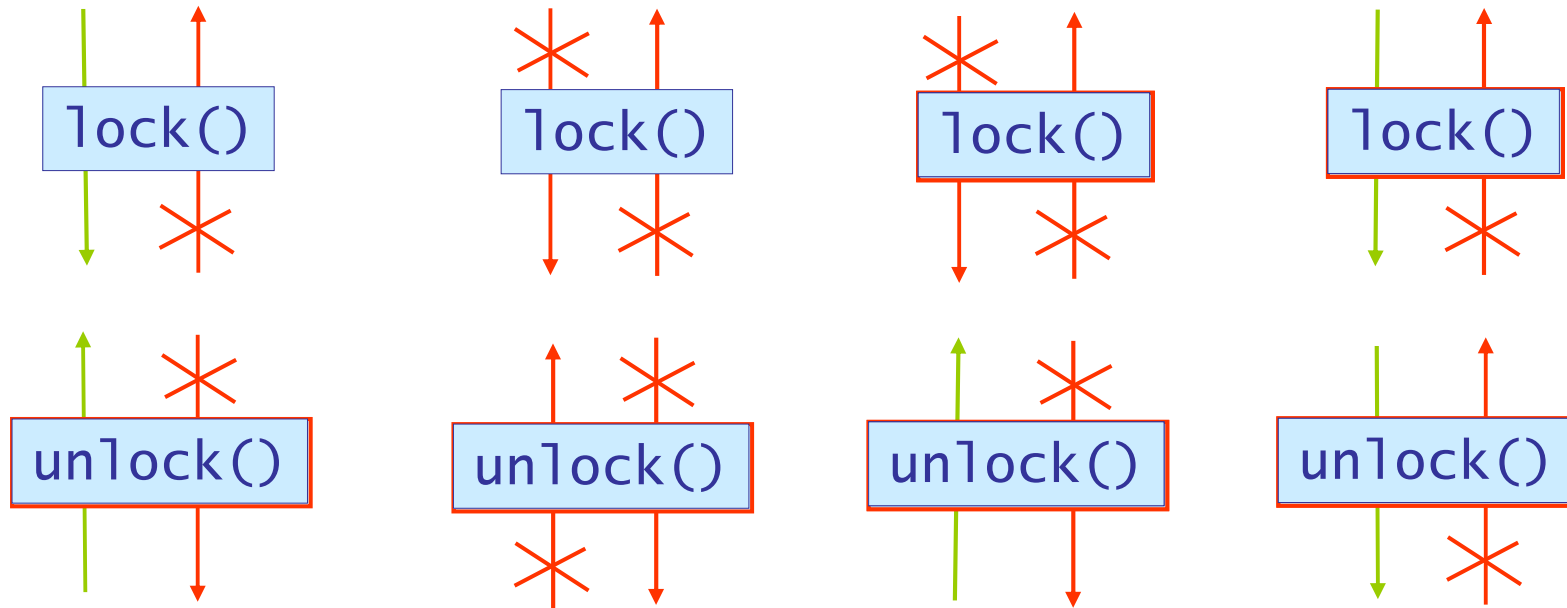
Naïve pthreads:



Optimized pthreads



Some open source pthread lock implementations (2006):



[technically incorrect]

NPTL

{Alpha, PowerPC}

{mutex, spin}

[Correct, slow]

NPTL

Itanium (&X86)

mutex

[Correct]

NPTL

{ Itanium, X86 }

spin

[Incorrect]

FreeBSD

Itanium

spin