Towards a Memory Model for C++

Hans-J. Boehm

HP Labs

Acknowledgments

- Some of this has benefited greatly from discussions with Bill Pugh, Doug Lea, Peter Dimov, Clark Nelson, Alexander Terekhov, Andrei Alexandrescu, Herb Sutter, David Callahan, Lawrence Crowl, Sarita Adve, Jeremy Manson, ….

Disclaimer

- None of this is even in the working draft yet.
- Anything may change.
  - I’m trying not to predict the outcome of the most controversial issues.

Rough Outline

1. What's wrong with the C/C++ & Pthreads/win32 programming model
   - And why we need a proper memory model to describe thread interaction.
2. What we are doing about it
   - General approach.
   - Impact on programming model.
   - Atomic operations.

Multi-threaded programming is increasingly important:

- It continues to be widely used to deal with multiple event streams.
- We need parallel programs to take advantage of multi-core processors.
  - And those are likely to be the main source of improved performance.
- Threads are the obvious way to get there.
  - The APIs exist and are widely used.
  - And sometimes only shared memory yields sufficient performance.

Common approaches:

- Threads integral to the language:
  - Java
    - Hard to get the semantics right.
    - See Manson, Pugh, Adve, “The Java Memory Model”, POPL05
    - Somewhat different concerns from C++.
  - C#, …
  - Not this talk.
- Single-threaded language + threads library.
  - C/C++ & Pthreads, Win32 threads, …
  - Simple: Compiler & language spec oblivious to threads.
  - This talk: Close, but inherently not quite correct.

We use pthreads for details

- The rules governing shared variable accesses are relatively well specified.
- Widely used, surprisingly close to “correct” for such a simple spec.
- Original win32 threads approach appears identical.
  - But I couldn’t find the analogous specification.
  - Differences in synchronization primitives not relevant.

Pthreads rules

No concurrent modification to shared variables (no races):

"Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. *[i.e. no data races.]* Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads…"

- Single Unix SPEC V3 & others

These functions include **`pthread_mutex_lock()`** …

- Seemingly independent of language specification.
- C and C++ specifications don't mention threads.

Why no data races?

- Almost dodges *memory model* issues:

(Initially x = y = 0)

Thread 1	*Thread 2*
x = 1;	y = 1;
r1 = y;	r2 = x;

Can r1 = r2 = 0?

  - Intuitively (or under sequential consistency) no; some thread executes first.
  - In practice, yes; compilers and hardware can reorder.

- Under pthreads rules this is simply illegal.
  - We don't really get to ask the question.

Reordering limitations

- Normally compiler/hardware can reorder instructions, so long as this is correct for 1 thread:

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

- But reordering around synchronization calls is bad:

```
pthread_mutex_lock(…);          pthread_mutex_lock(…);
x++;                        ⟶   pthread_mutex_unlock(…);
pthread_mutex_unlock(…);        x++;
```

- And implementations prevent this.

How Pthreads Implementations (and win32 threads?) *almost* work

- Synchronization-free code can be optimized as though it were single-threaded.
  - If a thread could observe the difference, the observer would introduce a race.
- Synchronization functions contain any needed hardware memory fences.
- Synchronization functions are treated as opaque by compilers.
  - The compiler views them as potentially reading or writing any global.
- Compilers can't normally move memory references across them.
- Compilers that follow single-threaded optimization rules *rarely* break multi-threaded code.

Why this doesn't quite work
… and threads need to be in the language

1. The basic rules are circular.
   - You need to know what programs mean to define "concurrent modification" and races.
2. What's a "memory location"?
   - The language spec must say.
   - Impacts compiler.
3. What does it mean to "synchronize memory"?
   - The straight-forward interpretation
     - Is easy to implement.
     - Can unexpectedly break code.
4. Performance for the 2% (??) of code that needs shared variable access without locks.
   - Which may affect overall performance substantially.

Why it matters

- The current specifications do produce rare failures in practice. (One example to follow.)
- Since no component is clearly violating specifications, these are hard to track down and fix.
  - Inconsistent/nonportable workarounds/fixes.
- It's very hard to state coding guidelines that avoid these problems.
- We need to teach more people to write correct multithreaded code.
  - That's harder if the rules don't make sense.

Concurrent modification (data races)

- Does the following program access a memory location "while another thread of control may be modifying it"?

Initially `x` = `y` = 0

Thread 1

```
If (x == 1) ++y;
```

Thread 2

```
if (y == 1) ++x;
```

or how about

```
++y; if (x != 1) --y;
```

```
++x; if (y != 1) –-x;
```

?

- Need a semantics for the concurrent language to define when there is a race.

“Memory location”:

- Consider

```
x = y = 0
```

Thread 1	*Thread 2*
`x = 1;`	`y = 1;`

- Can we guarantee x = y = 1 when both threads finish?
  - If x and y are adjacent bit-fields, no.
  - If x and y are adjacent char members on an ancient Alpha, no.
  - If x and y are adjacent char members on a recent Alpha, compiled for backwards compatibility, no.
  - If x and y are “close” data members, and the compiler decided to combine the store with another one, maybe not.
  - If one of x or y is a bit-field, and the other is close, maybe not .
- Pthread’s uniform answer: No. They may occupy the same “memory location”.
- There is no portable pthreads code. Needs fix.

"Synchronize memory"?

- Really not sufficient: (register promotion)

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

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

“Synchronize memory” (contd.)

- Memory contents at lock() and unlock() calls reflect logical state.
  - Clearly not sufficient.
- Again compiler adds stores (and introduces races) not present in the source.
  - Invisible with single thread.
  - Visible to another thread.
  - Even gcc –O2 performs this kind of unsafe optimization.
  - Usually it doesn’t break anything.
- Very difficult to guard against.
- Language definition, compiler must preclude this.

Performance

- So far we were dealing with correctness.
- Pthreads rules require “fully synchronized” programs.
- A good idea for 98% of code.
- The following discussion is about the other 2%
  - … which may account for > 50% of application performance.

Fully synchronized programs can be slow

- Traditional pthread_mutex's require:
  - Dynamic library call
  - 2 x (Atomic op (e.g. CAS) + memory fence(s))
- Sample cycle costs:

	CAS	fence	lock/unl.
2.0 Xeon	124	125*	336
1.0 Itan.	10	4+	109
500 PIII	25	19*	156

*Not needed with compare-and-swap (CAS)

Faster alternatives:

- Extensive literature on lock-free algorithms.
- Here we use two examples:
  - "Double-checked" locking idiom.
  - Parallel GC with
    - locked mark bit update, vs.
    - "cheating" and relying on hardware facilities
      - In this case atomic byte stores.
      - Concurrent stores are safe in this case.
- My 2005 PLDI paper has Sieve of Eratosthenes example.

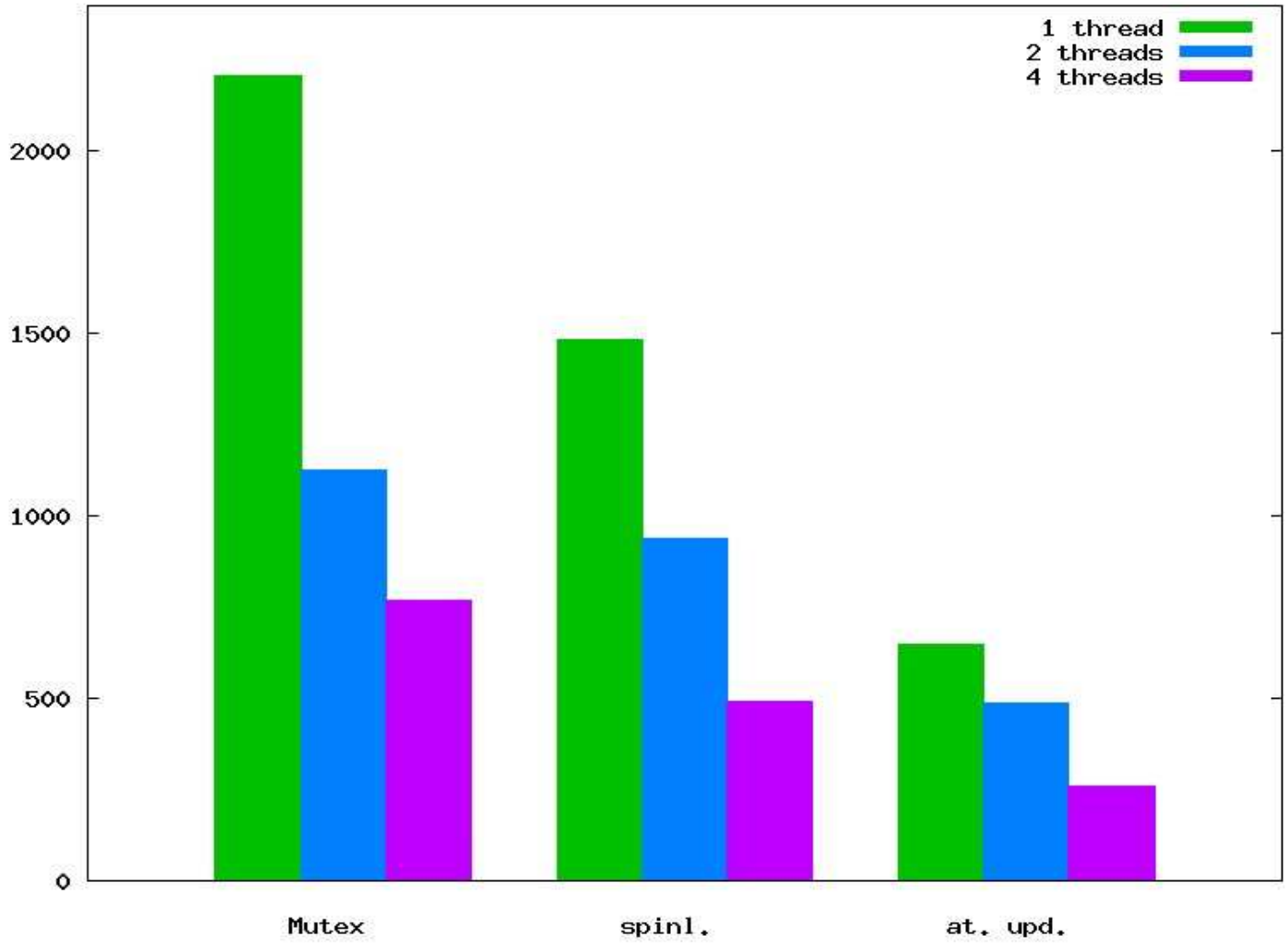
A pervasive example: Double-checked locking

- Assume we want to lazily initialize x
  - while avoiding locking on the fast path

```
if (!x_initialized) {
  lock();
  if (!x_initialized) x = …;
  x_initialized = true;
  unlock();
}
… x …
```

- Incorrect as is:
  - Data race on `x_initialized`!
  - May fail in practice for multiple reasons, esp. on some hardware.
- But we want something like this to work.

P4 parallel GC trace performance (Time to trace 200 MB, msecs)



Parallel GC performance summary

- `Pthread_mutex_lock()` version on 4 “processors” (2x2 threads) is slower than uniprocessor version.
- Why bother with a multiprocessor?
- Sometimes concurrent writes to shared variables are unavoidable.

The bottom line:

- Language specification must address threads.
  - When can there be a data race?
  - When can adjacent data be rewritten as part of an assignment?
  - Can additional reads and writes of globals be introduced by the compiler?
  - Any guarantees with data races?
  - These are all compiler/language issues.
- Need atomic operations support for occasional un-locked access to shared globals.

Solution for C++0x:

- We're defining a "memory model" or thread semantics describing visibility of memory accesses to other threads, and answering those questions.
  - Java now has a reasonable one (Pugh, Manson, Adve, others)
  - We're working on C++.
    - Also: Andrei Alexandrescu, Doug Lea, Bill Pugh, Clark Nelson, Maged Michael, Ben Hutchings, Peter Dimov, Alexander Terekhov, Sarita Adve, Lawrence Crowl, and others.
    - Herb Sutter is pursuing similar goals for Microsoft (not .NET)
    - Rest of this talk.

Observation:

- Type-safe languages supporting sandboxing have relatively strong requirements:
  - Completely undefined semantics, even for bad code, are generally not acceptable.
    - Malicious code may exercise undefined semantics.
    - We do care what malicious code can do.
  - Some guarantees have to apply even to misbehaved code.
- Those requirements have a performance cost.
- And they seem to add complexity to the specification.

C++ constraints are a bit different:

- Undefined semantics are tolerable.
- Java-style rules would probably be more expensive than in Java.
  - Would add overhead for object construction.

Our Approach:

- "Pthreads-like" memory model.
- Data races (updates to a memory location concurrent with another access) have undefined semantics.
- Corollary: *There are no benign data races*.
  - We'll give you atomic objects you can use instead.
- Otherwise: Formally similar to Java model.
  - Sequential consistency with simple locks.
- Careful and restrictive definition of "memory location" and "data race".
  - A "memory location" is any non-bit-field object, or contiguous sequence of (!= 0 length) bit-fields declared in the same structure.
- Implicitly, extra stores cannot be visibly introduced.

The big picture: From the user's perspective

- In 98% of cases, basic methodology isn't changing:
  - Acquire locks to protect against data races.
  - Hold locks long enough to ensure required atomicity.
  - Generally applies at the level of scalar objects, structs, STL containers, etc.
    - Containers generally acquire locks only to protect hidden shared state.
    - It is the client's responsibility to protect against simultaneous accesses to a container, if at least one access is a write.
    - I/O?
  - "`volatile`" will probably still have nothing to do with threads.
- C++0x will provide a standard threads API
  - But it should look fairly familiar.
- May eventually move to transactional memory
  - but not yet…

What is changing:

- The rules are becoming clear.
- If your code doesn't use atomics(?), abuse try_lock, or contain (certain kinds of) infinite loops:
  - You can determine whether your program contains a race by viewing execution as an interleaving of thread actions (sequentially consistent semantics).
  - If your program does not contain a race, it executes as if thread actions were interleaved (sequential consistency).
  - The compiler does not introduce data races, except that an assignment to a bit-field is viewed as modifying the whole contiguous sequence of bit-fields for race determination.

These rules are not always followed by current compilers, e.g.:

- Stores to struct/class members may not unnecessarily overwrite adjacent members.
  - Intel Example:
    - `struct {char a; int b:9; int c:7; char d;}`
  - A store to `b` must be implemented as 2 separate 1-byte stores.
  - No production compilers (?) currently do this.
- Speculative register promotion often illegal:

```
for (T *p = q; p != 0; p = p -> next)
  if (p -> data > 0) ++count;
```

  - Standard register promotion of `count` becomes illegal.
  - Gcc can violate this rule.

Atomic operations add complications

- We need atomic operations which support concurrent access w/o locking.
  - For sophisticated lock-free data structures.
  - For a few idioms like double-checked locking.
- This complicates the definition of data race
- … and potentially raises a lot of the (hard) Java issues
- … and then some
  - Because we want more general atomic operations.
- Most of the technical challenges reside here.
- But they should be rarely used.

Atomic operations (contd.)

- Current thinking is to provide two levels:
  - Java-volatile-like high level.
    - Normal assignment syntax for atomic<T> variables.
    - May provide sequential consistency properties similar to locks.
      - Ongoing negotiations: feasibility is unclear.
  - Also lower level operations with finer control.
  - Here we concentrate on the low-level
    - Since that exposes the complications.
    - And explains why you probably don't want to use it!

Atomic operations need ordering constraints

- Reconsider double-checked locking:

```
if (!x_initialized) {
  lock();
  if (!x_initialized) x = …;
  x_initialized = true;
  unlock();
}
… x …
```

- Data race on `x_initialized`!
  - Fix with
    - `atomic<bool> x_initialized;`
  - Races on atomic variables are OK, but …

Ordering constraints contd.

- Compiler transformations can still break things:

```
tmp_x = x;
if (!x_initialized) {
  lock();
  if (!x_initialized) x = ...;
  x_initialized = true;
  unlock();
}
... tmp_x ...
```

- `x` read before `x_initialized`!
- Need to limit compiler reordering around atomic operations, in addition to locks.

Ordering constraints (contd 2)

- We really want:
  - Memory operations preceding an atomic store to become visible before the store.
  - Operations performed after an atomic load to become visible after the load.
  - Effectively the atomic write to `x_initialized` makes prior memory operations visible to another thread that reads `x_initialized`.
  - For memory visibility, the write to `x_initialized` behaves like the release of a lock, and the read behaves like a lock acquisition.

Ordering constraints (contd 3)

- For low-level atomics, we explicitly indicate these ordering constraints with load_acquire, store_release.
- These inhibit compiler & hardware reordering.
  - May require expensive memory fence instructions.
- Real syntax and some details TBD.

Double-checked locking:
Correct, with low-level atomic operations

- Using obsolete(?), intentionally ugly, very explicit, syntax:

```
atomic<bool> x_initialized;
                //Only required change

if (!x_initialized.load_acquire()) {
  lock();
  if (!x_initialized.load_relaxed())
    x = ...;
  x_initialized.store_release(true);
  unlock();
}
... x ...
```

Memory model treatment of atomics (50,000 foot view)

- Mostly standard, similar to Java.
- Define “happens-before” visibility ordering.
- A read *r* may not see a write *w* if
  - *r* happens before w, or
  - another write “happens between” them.

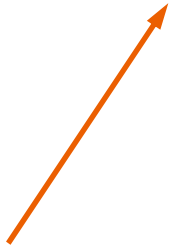
Happens-before illustration (atomics)

Thread 1

```
if (!x_init.ld_acq())
{
   lock();
   if (!x_init.ld…())
     x = ...;
   x_init.store_rel(1);
   unlock();
}
... x ...
```

Thread 2

```
if (!x_init.ld_acq())
{
   lock();
   if (!x_init.ld…())
     x = ...;
   x_init.store_rel(1);
   unlock();
}
... x ...
```



Low-level atomics are still hard to use

- Reconsider original example variant:

(Initially x = y = 0)

Thread 1	Thread 2
x.store_release(1);	y.store_release(1);
r1 = y.load_acquire();	r2 = x.load_acquire();

Can r1 = r2 = 0?

- Yes!
  - None of our previous constraints apply here.
  - Doesn't matter for some comon cases, but Dekker's algorithm breaks, etc.
  - High level atomics also prohibit this, simplify syntax.

Current status

- The C++0x registration draft includes place holders for memory model, other thread support.
- Web page at

  http://www.hpl.hp.com/personal/Hans_Boehm/c++mm
- Includes pointers to formal memory model proposal (with Clark Nelson(Intel)), and more detailed explanations.
- Preliminary atomic operations library interface (with Lawrence Crowl, help from Peter Dimov).
  - Converging but we're still iterating.
- An official C++ threads API is also progressing …
- The C committee has been watching, and may follow.

Backup slides

Speculative register promotion, again

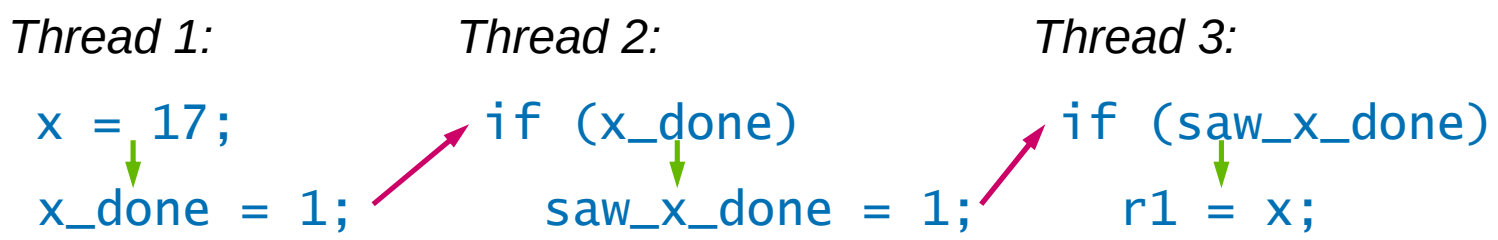
- Note that even very simple cases can be unsafe:

```
[ count is global ]
for (p = q; p != 0; p = p -> next) {
    if (p -> val > 0.0) count++;
}
```

- May not touch `count` if `q` has only negative entries.
- Promoting `count` to a register introduces unconditional access, but
- Unconditionally setting global `count` at the end of the loop may introduce race and is unsafe!
- Even gcc –O2 does this.

Architectural implications

- Byte stores are basically required.
- Atomic operations (e.g. cmpxchg) highly desirable.
- Need a cheap way to enforce transitivity of happens-before (basically causal ordering):
  - Assume x_done, saw_x_done are atomic:



•As with Java volatiles, `r1` must be 17.

Double-checked locking: Why it is prohibited by pthreads.

- Compiler/hardware may reorder

```
if (!x_initialized) {
  lock(); // Not real syntax
  if (!x_initialized) x = ...;
  x_initialized = true;
  unlock();
}
... x ...
```

- E.g., compiler may load `x` early after discovering that it misses cache.
- Some architectures allow hardware reordering of loads of `x` and `x_initialized`.
- In this case, we also need to restrict reordering.

Faster alternatives: A Simple Example

- Sieve of Eratosthenes
  - Compute primes between 10K and 100M
  - Each thread executes:

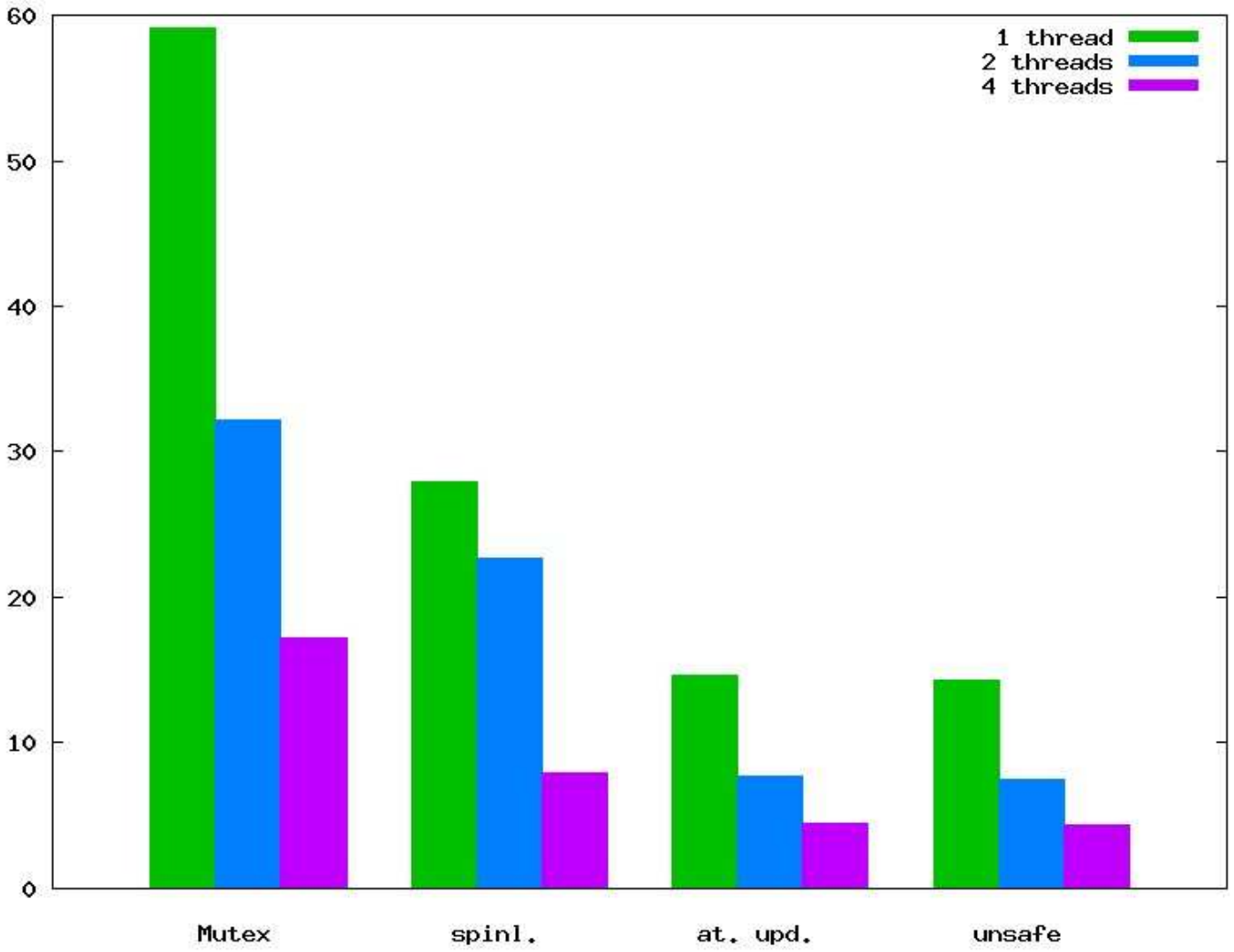
```
for (my_prime = start; my_prime < 10000; ++my_prime)
    if (!get(my_prime)) {
        for (multiple = my_prime; multiple < 100000000;
             multiple += my_prime)
            if (!get(multiple)) set(multiple);
    }
```

  - Get/set operate on 100M “bit” array.
  - Similar to core of some parallel garbage collectors.

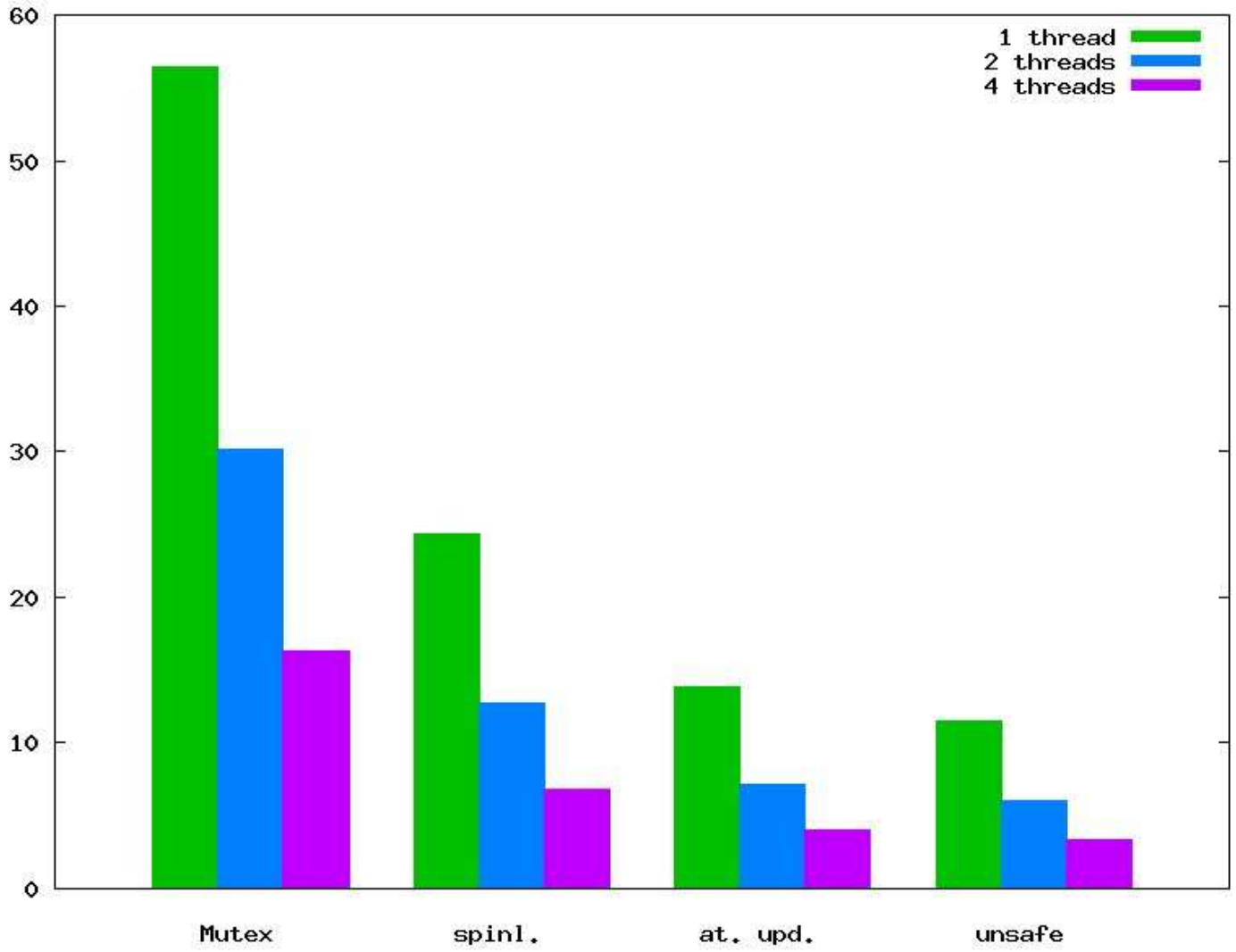
Measurements: Alternatives

- Bit array vs. byte array
- Synchronization alternatives:
  - None (thread unsafe, but usually “works”)
  - Atomic byte stores for bytes
  - Atomic or into bit array
    - Needs portable access to e.g. cas
  - **`Pthread_mutex`** locks (every 256 bits)
  - **`Pthread_spin`** locks (every 256 bits)

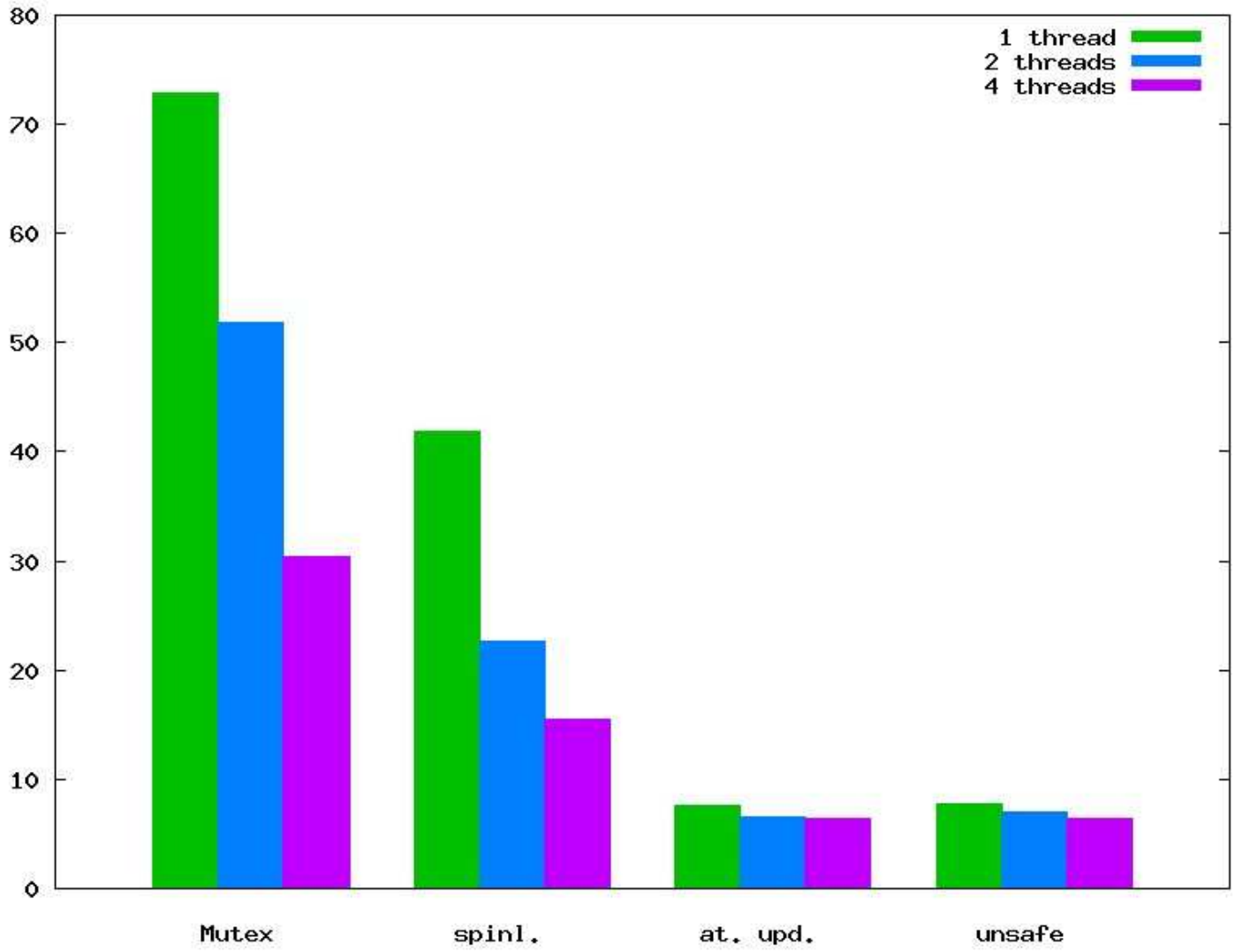
Itanium (gcc) running time 4x1GHz, bytes



Itanium (gcc) running time 4x1GHz, bits



Pentium 4 time (2x HT 2GHz), bytes



Performance summary

- **`Pthread_mutex_lock()`** version on 4 “processors” is slower than uniprocessor version.
  - Why bother with a multiprocessor?
- Versions with atomic operations or byte arrays either
  - Scale reasonably, or
  - Saturate the bus/memory (?)
- In the first case
  - We can get good speedups even in this contrived case.
  - Real code benefits substantially from atomic operation access.
- In the second case
  - Contrived case doesn’t speed up.
  - Real code requires atomic operations for speed up.