



# Towards a Memory Model for C++

**Hans-J. Boehm**  
**HP Laboratories**

© 2004 Hewlett-Packard Development Company, L.P.  
The information contained herein is subject to change without notice



# Acknowledgments

- Some of this is joint work with Bill Pugh, Doug Lea, Peter Dimov, Clark Nelson, Alexander Terekhov, Andrei Alexandrescu, ....
- Many of the observations here, e.g. the difficulty in defining a “memory location” have been known for a long time, but seemed to be underappreciated.

# 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
  - Overview of the technical challenges.
  - Why atomic operations matter.
  - Our approach.
  - Implementation consequences.

# 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
  - C#, ...
- Single-threaded language + threads library.
  - Assume no type-safety/sandboxing concerns.
  - 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.
- Java/C# present different issues

# 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

$x = 1;$

$r1 = y;$

Thread 2

$y = 1;$

$r2 = x;$

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.



# 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 barriers.
- 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 a memory model 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.

# Concurrent modification

- 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;
```

?

# What's missing?

- Must define which reads and writes can occur.
  - Requires a semantics for the concurrent language.
  - Requires a “memory model” that defines allowable reordering.
  - But that's exactly what library-based threads packages try to avoid.
- Java makes strong guarantees sequential consistency for programs that are data-race free *in sequentially consistent executions*.
- We will assert that C++ should make a similar guarantee.

# What's a "memory location"?

- Consider `struct {t1 f; t2 g;}`
- When is it legal to concurrently write `x.f` and `x.g`?
  - It isn't if f and g are adjacent 1-bit bit-fields.
    - Writing one reads and rewrites the other.
  - It isn't for adjacent `char` fields on an early Alpha machine.
  - Nearly every program does so for some fields.
- Similar issues:
  - Sometimes compilers optimize by reading and rewriting adjacent fields
  - Simultaneous byte array writes.
  - "Close" global variables.
- Posix intentionally does not specify when a memory location is shared.

## “Memory location” contd.

- Strict interpretation of Posix quickly becomes intolerable:

```
char x = 0, y = 0;
```

Thread 1

```
x = 1;
```

Thread 2

```
y = 1;
```

Either x or y may be zero (or 42?) after both threads finish!

- This is independent of declaration order.
- There is no portable pthreads code.
- Appears to have an easy fix.
  - Assuming modern hardware.

# "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.
  - This is
    - Unacceptable.
    - Common optimization for both commercial and research compilers.
    - Rarely visible, hard to test.
- Language definition, compiler must preclude this.

# 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.

# Performance

- So far we were dealing with correctness.
- Pthreads rules require “fully synchronized” programs.
- A good idea for 98% of code.
- The rest of this 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 + memory barrier(s))
- Sample cycle costs:

	CAS	barrier	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:

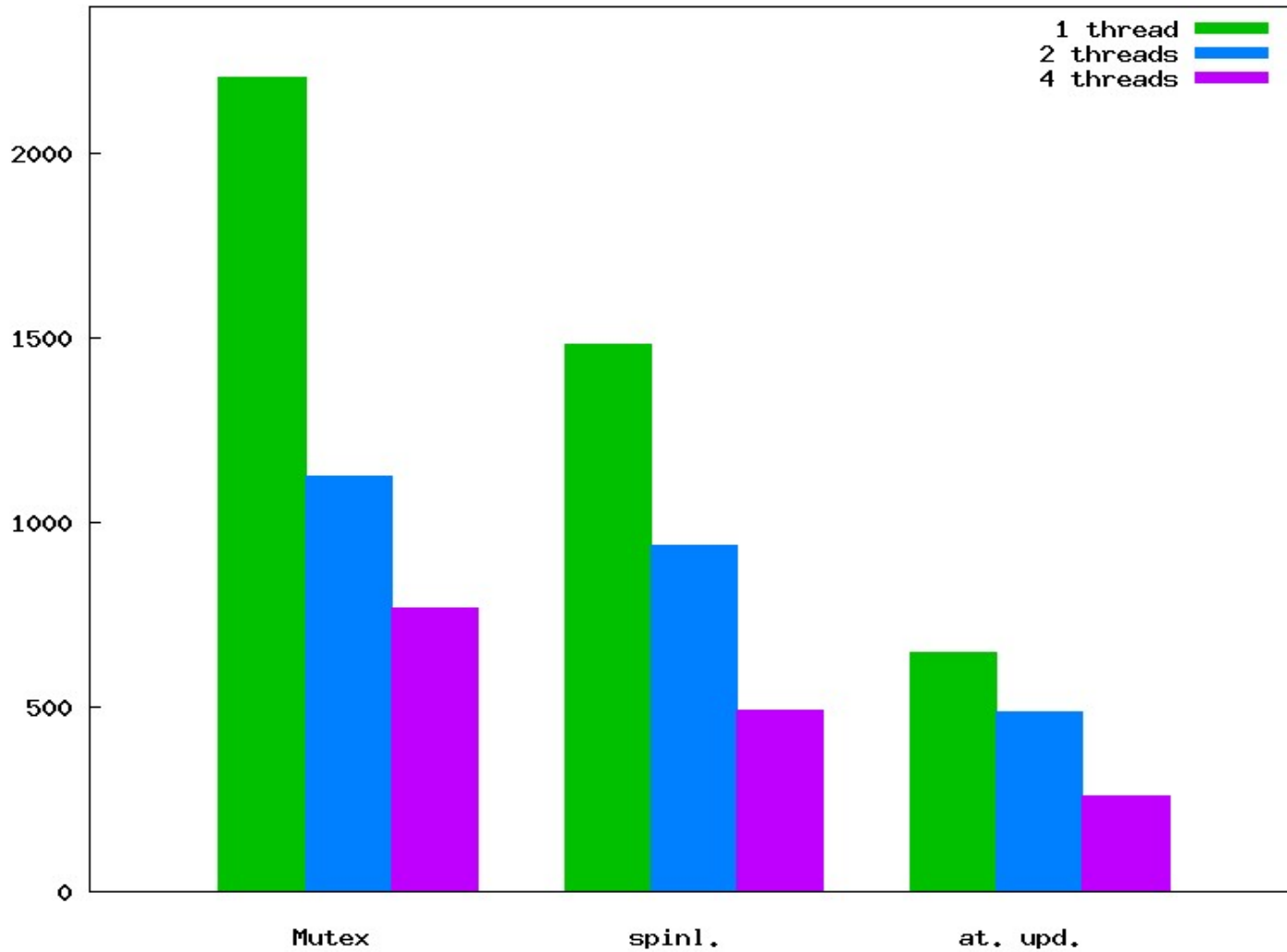
- Extensive literature on lock-free algorithms.
- Here we use two examples:
  - “Double-checked” locking idiom.
  - Parallel GC with
    - locked mark bit update, vs.
    - “cheating” and relying on hardware facilities
      - In this case atomic byte stores.
      - Concurrent stores are safe in this case.
- 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, msec)



# 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.

# What does all of this mean?

- Pthread-like thread specifications are inadequate.
  - Language specification must be involved.
    - 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 unlocked access to shared globals.
  - A proper memory model for the language makes all of this feasible.

# What are we doing about this?

- We need a “memory model” 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 and others.
    - Currently we have strong support from C++ committee.
    - Different Technical challenges.
    - Rest of this talk.

# Observation:

- Type-safe languages supporting sandboxing have relatively strong requirements:
  - Completely undefined semantics are generally not acceptable.
    - Malicious code may exercise undefined semantics.
    - We do care what malicious code can do.
  - Need some immutability guarantees.
    - Otherwise security checks involving e.g. strings are meaningless.
- Those requirements seem to add complexity.
  - Something like Java's causality treatment appears hard to avoid.

# C++ constraints are different:

- Undefined semantics are OK.
- Fully defined semantics are relatively more expensive?

```
foo *y = 0;
```

```
Thread 1: foo x(); y = &x;
```

```
Thread 2: if (y != 0) y -> bar();
```

– Is thread 2 allowed to see a partially initialized pointer, or a bad vtable pointer and take a wild branch?

- Disallowing this requires:
  - Atomic pointer assignments (even on small embedded processors).
  - Potentially memory fences at the end of constructors.
    - And those are more frequent than in Java.

# Our Approach:



- "Pthreads-like" memory model.
- Data races (updates to a memory location concurrent with another access) have undefined semantics.
- Otherwise: Sequential consistency.
- Careful and restrictive definition of "memory location" and "data race".
- Only bit-fields share a "memory location."

# But atomic operations add complications



- We need atomic operations which support concurrent access.
  - 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 (?)

# Atomic operations:

Allow concurrent access in special situations

- Such as double-checked locking:
  - Loads and stores of `x_initialized` must be done specially:
    - Tell compiler (and programmer) that a race is involved.
    - Ensure atomicity.
    - Specify ordering constraints.

# Double-checked locking: Correct, with atomic operations

- Using tentative, needlessly explicit, syntax:

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

```
if (!x_initialized.load<acquire>()) {  
    lock();  
    if (!x_initialized.load<raw>())  
        x = ...;  
    x_initialized.store<release>(true);  
    unlock();  
}  
... x ...
```

- **store<release>** ensures that preceding stores are visible to a **load<acquire>** reading variable in another thread.
  - Allow concurrent access
  - Inhibit compiler, hardware reordering

# Atomic operation semantics

Variables `x`, `y` and `z` initially 0

## Thread 1:

```
x.store<raw>(1);  
r1 = y.load<raw>();  
if (r1 == 0) z = 17;
```

## Thread 2:

```
y.store<raw>(1);  
r2 = x.load<raw>();  
if (r2 == 0) z = 42;
```

- Does this have a data race?
- Simultaneous accesses through atomics don't count.
- No race on `z` under sequentially consistent interpretation.
- But simultaneous accesses are really possible.
- This must have undefined semantics in order to preserve the compilers optimization ability.
- But how do we specify this? While dodging Java issues?

# Our approach



- Define a “happens-before” relation to take the place of sequential ordering for update visibility.
  - As usual, not a total order.
- Happens-before is basically
  - intra-thread sequencing for non-atomic operations +
  - release op happens-before next acquire of same lock or atomic object +
  - some intra-thread ordering for atomics, locks.
    - (Some complexity here, but can usually be ignored.)
- Object reads do not see assignments that “happen-after” them, or that are hidden by assignments that “happen in-between”.
- If two accesses ( $> 1$  write) to the same non-atomic “memory location” occur without a “happens-before” ordering, there is a data race, and meaning is undefined.
  - Otherwise reads are allowed to see any value consistent with above.

# Happens-before illustration

Thread 1

`x = 0;`



`x = 1;`



`r1 = y;`

Thread 2

`y = 0;`



`y = 1;`



`r2 = x;`

- `r1 = r2 = 0` would be possible, but
- No happens-before ordering on stores & loads to `x` and `y` → race!

# Happens-before illustration (atomics)

Thread 1

`x = 0;`



`x = 1;`

`r1 = y;`

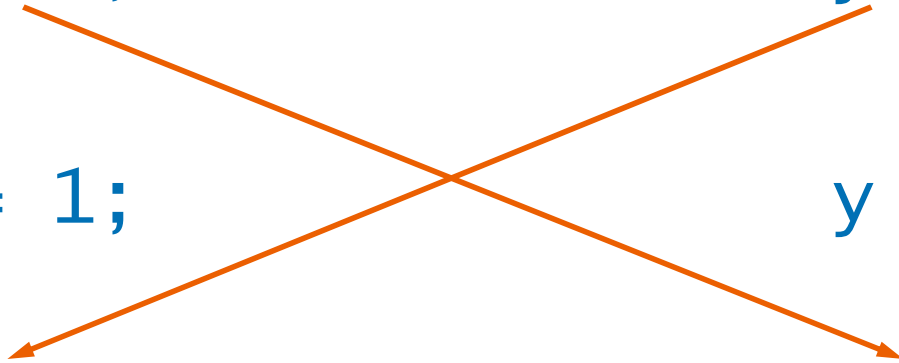
Thread 2

`y = 0;`



`y = 1;`

`r2 = x;`



- `r1 = r2 = 0` possible (unlike Java)
- No race, since these are atomics.

# Current approach continued

- A “memory location” is any non-bit-field object, or contiguous sequence of ( $\neq 0$  length) bit-fields declared in the same structure.
- Implicitly, extra stores cannot be visibly introduced.
- Claim: This guarantees sequential consistency for race-free programs using locks for synchronization.
  - (If you use `tryLock()`, this is true, but only because our definition of data race is different from Posix’.)

# Does require compiler changes:

- No speculative or unnecessary stores.
  - 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 byte stores.
    - 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.
    - Some kinds of code hoisting are problematic.
    - Stores may not be moved above potentially non-terminating loops.

# 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).

# Current status



- Web page at [http://www.hpl.hp.com/personal/Hans\\_Boehm/c++mm](http://www.hpl.hp.com/personal/Hans_Boehm/c++mm)
- Includes (still informal) proposal, pointers to standards committee papers, archives, ...
- Needs further scrutiny.
- Very preliminary atomic operations library interface.
  - Details a bit controversial.



# Backup slides

# 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:

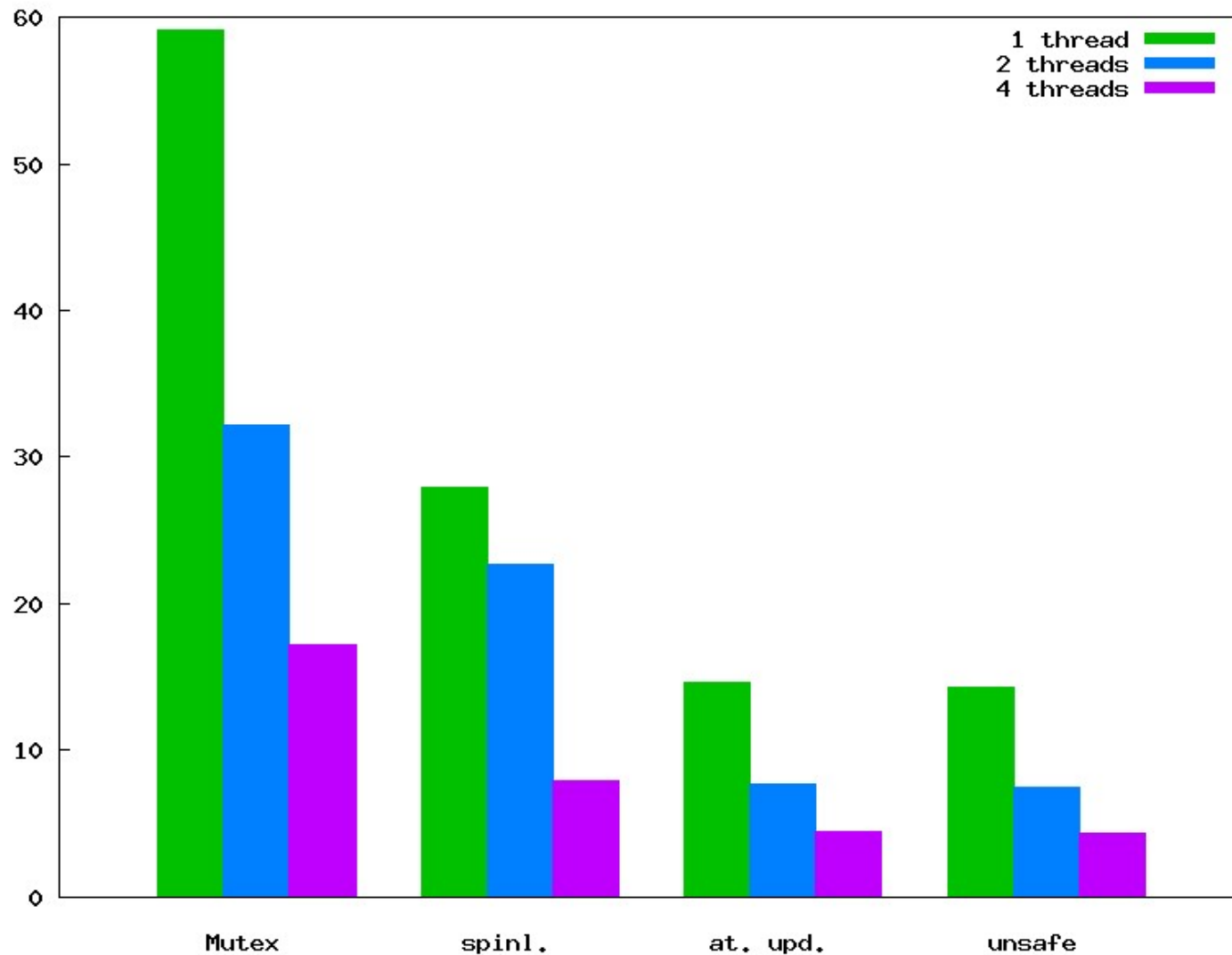
```
for (my_prime = start; my_prime < 10000; ++my_prime)
  if (!get(my_prime)) {
    for (multiple = my_prime; multiple < 100000000;
        multiple += my_prime)
      if (!get(multiple)) set(multiple);
  }
```

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

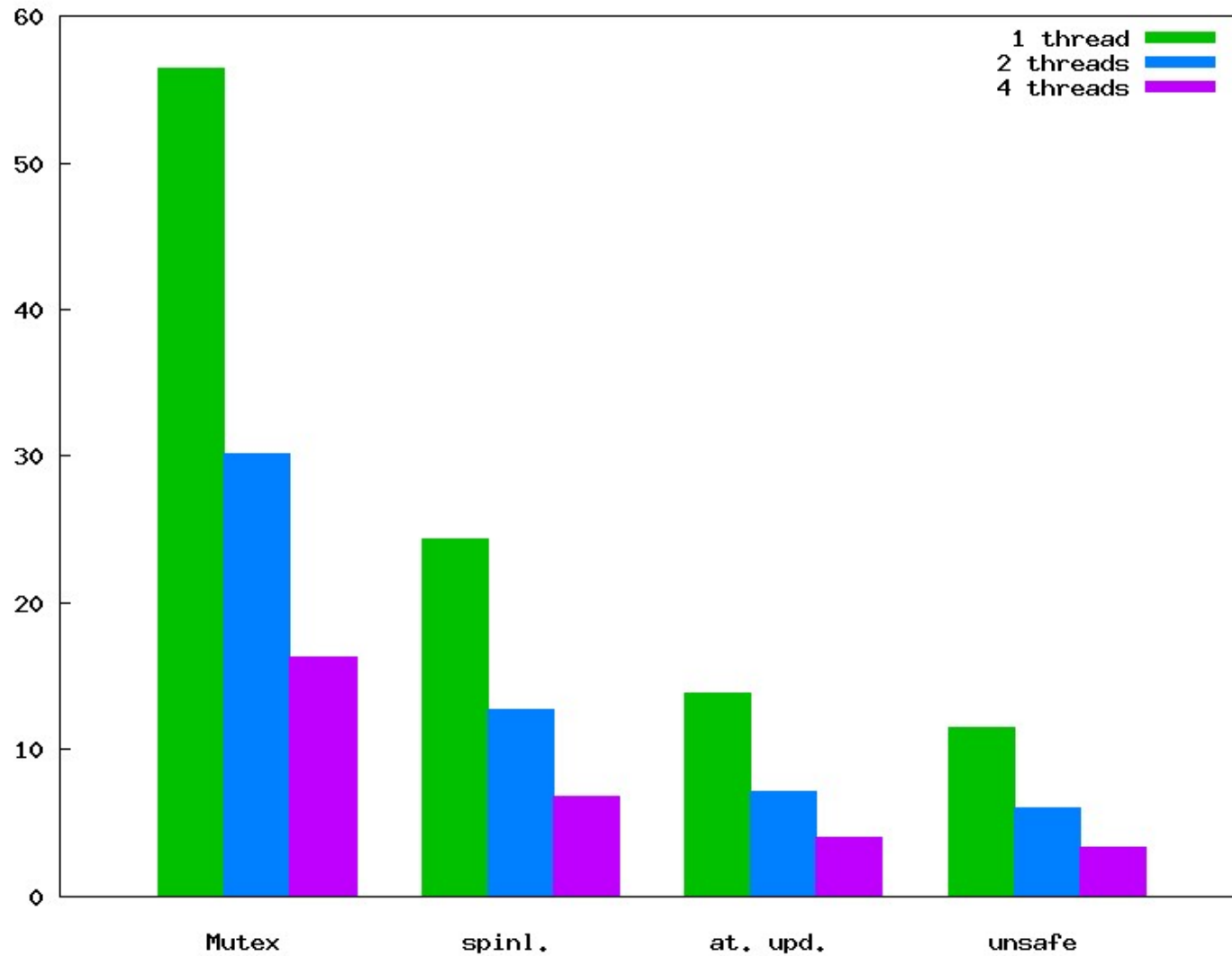
## Measurements: Alternatives

- 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)

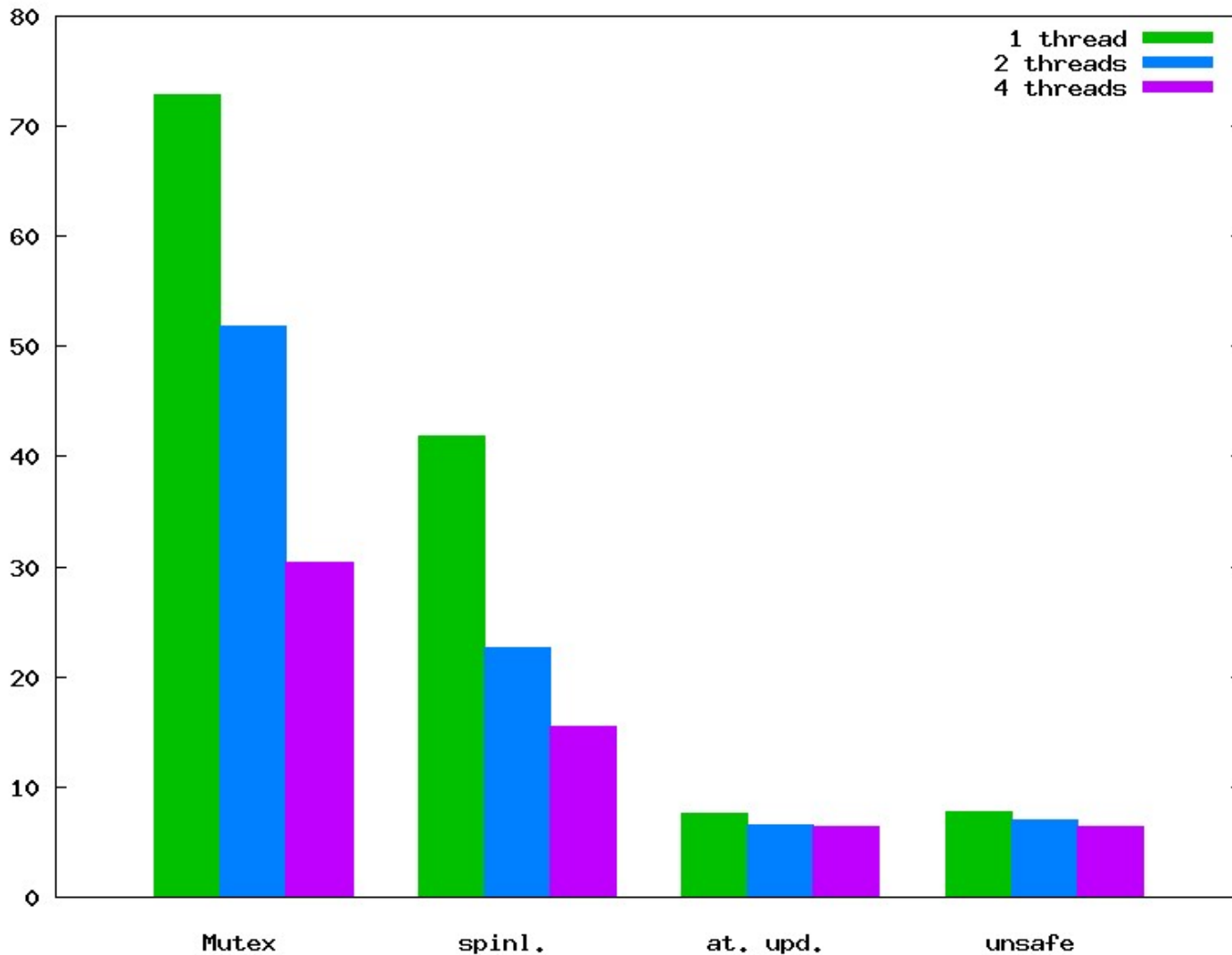
# Itanium (gcc) running time 4x1GHz, bytes



# 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.