Threads Cannot be Implemented as a Library

Hans-J. Boehm
HP Laboratories

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

Why multi-threaded programming?

- For program structuring.
  - Nothing changed for at least 10 years.
- For performance on multiprocessors.
  - As long as I remember, we claimed that multiprocessors were going to be the future.
  - They finally arrived in the mainstream:
    - Hardware multi-threading to hide memory latencies.
      - In half of PCs?
      - Logically another processor.
    - Multicore chips.
      - Starting to arrive in desktop PCs.
    - Both are likely to stay.
- Performance competitive code will increasingly have to be (preemptively) multi-threaded.

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 paper: Close, but inherently not quite correct.
- We pick on Pthreads because:
  - It has a relatively clean, carefully written, specification.
  - Widely used, surprisingly close to "correct".

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. 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 concurrent variable access?

- 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 no; some thread executes first.
  - In practice, yes; compilers and hardware can reorder.
- Under pthreads rules this is simply illegal.

How Pthreads Implementations *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 Pthreads don’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.
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.
     - Unexpectedly breaks code.
4. Performance for the 2% (??) of code that needs shared variable access without locks.
   - Which may affect overall performance substantially.

- Except for (3) & maybe (1), this was “known”, not widely appreciated.

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;
```

?

How do you fix the definition?

- 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*.
- Pthreads/C/C++ should make a similar guarantee.
- Are compilers still correct w.r.t. that interpretation?

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 quickly becomes intolerable:

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

Thread 1                Thread 2
x = 1;                  w = 1;
y = 1;
z = 1;
```

  Any variable may be zero 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) {
    count++;
}
```

- May be a concurrent update of count if q == 0.
- Unconditionally setting global `count` at the end of the loop 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:

- Examples:
  - Reference counts (atomic add).
    - C++ standard libraries not fully synchronized.
  - "Or" into "bit" vector.
    - Atomic byte store.
    - Atomic-or into memory location.
    - Eratosthenes Sieve example in paper.
  - Hash table or cache read without locking.
  - “Double-checked” locking.
    - Though that’s rarely done correctly.
  - Extensive literature on lock-free algorithms.

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

1 thread
2 threads
4 threads
2000
1500
1000
500
0
Mutex
spinl.
at. upd.

Performance summary

- `Pthread_mutex_lock()` version on 4 "processors" 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.
  - Specification must involve the language specification.
    - When can there be a data race?
      - Only if it occurs in the sequentially consistent interpretation(?)
      - More complicated with low-level atomic operations.
    - When can adjacent data be rewritten as part of an assignment?
      - Only within an adjacent group of bit-fields.
    - Can additional reads and writes of globals be introduced by the compiler?
      - No, except duplicate accesses between synchronization points.
  - Need occasional un-locked access to shared globals.
    - A proper memory model makes that feasible.

What are we doing about this?

- We need a “memory model” describing visibility of memory accesses to other threads.
  - Java now has a reasonable one (Pugh, Manson, Adve, others)
    - Somewhat different issues. (Type safety, security influences)
  - We’re working on C++.
    - Also: Andrei Alexandrescu, Doug Lea, Bill Pugh, Maged Michael, Ben Hutchings, Peter Dimov, Alexander Terekhov and others.
    - http://www.hpl.hp.com/personal/Hans_Boehm/c++mm

Erratum (Bad conjecture in paper)

- Pthread_mutex_lock() needs more than “acquire” ordering semantics.

Backup slides

Atomic_ops example: Correct lazy initialization

```
if (!AO_load_acquire(&is_initialized)) {
   // Lock and recheck if unsafe
   // to reinitialize.
   object_to_initialize = initial_value;
   AO_store_release_write
                    (&is_initialized, 1);
}
// safe to access object_to_initialize here.
```

- Generates barriers on Alpha, ld.acq, st.rel on Itanium, compile-only barrier on X86.

Faster alternatives: A Simple Example

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

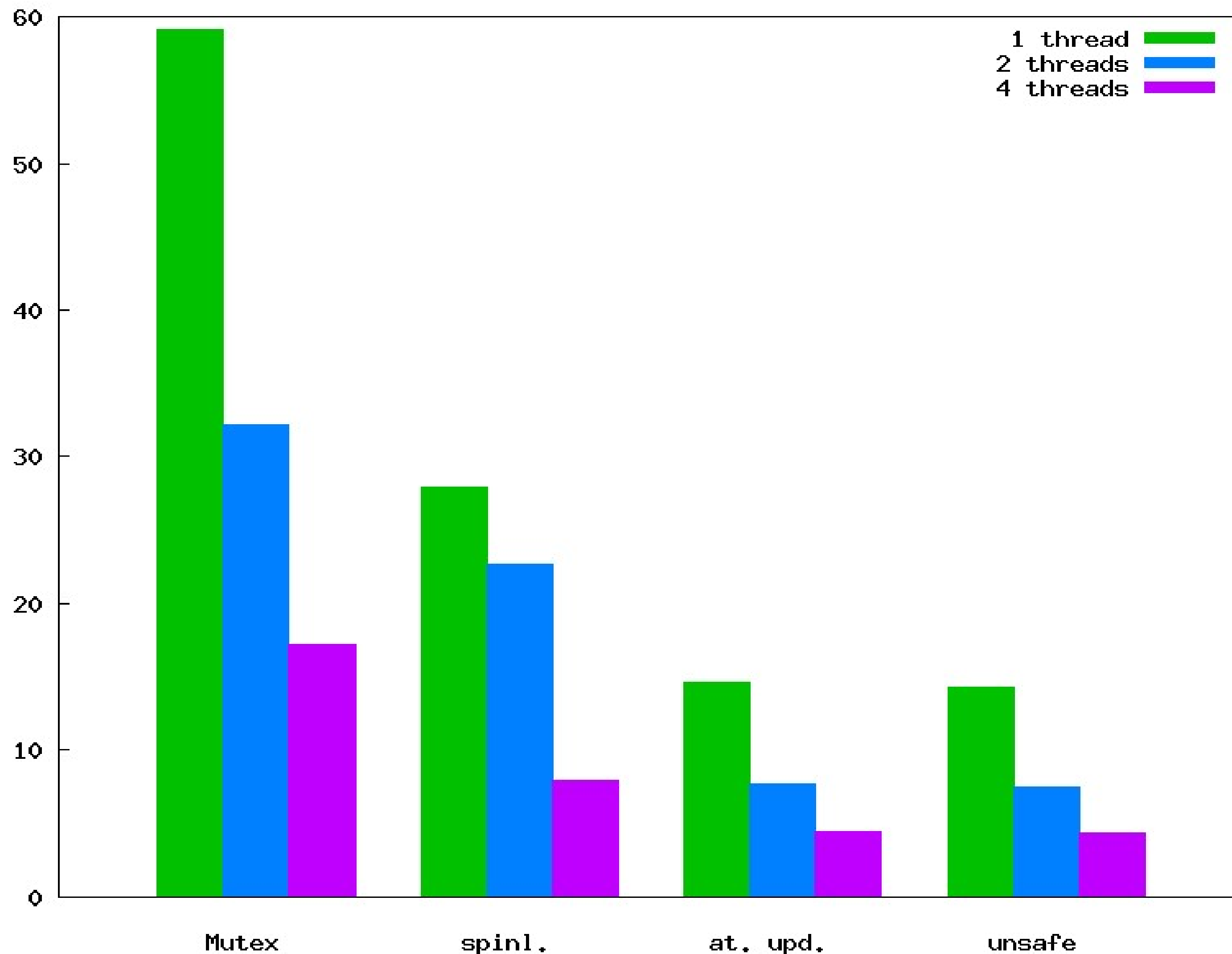
```
for (my_prime = start; my_prime < 10000; ++my_prime)
    if (!get(my_prime)) {
        for (multiple = my_prime; multiple < 100000000;
             multiple += my_prime)
            if (!get(multiple)) set(multiple);
    }
```

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

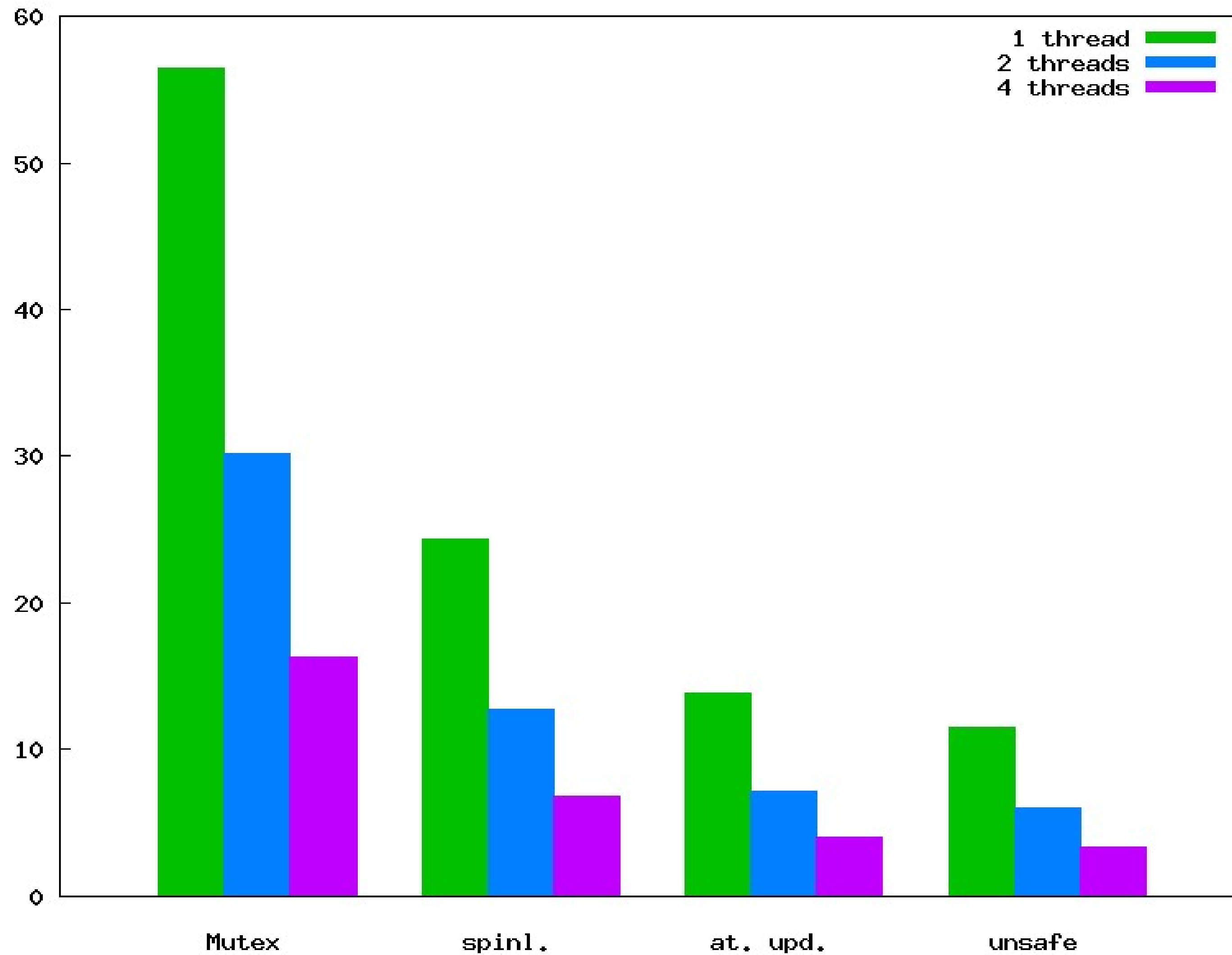
Measurements: Alternatives

- 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

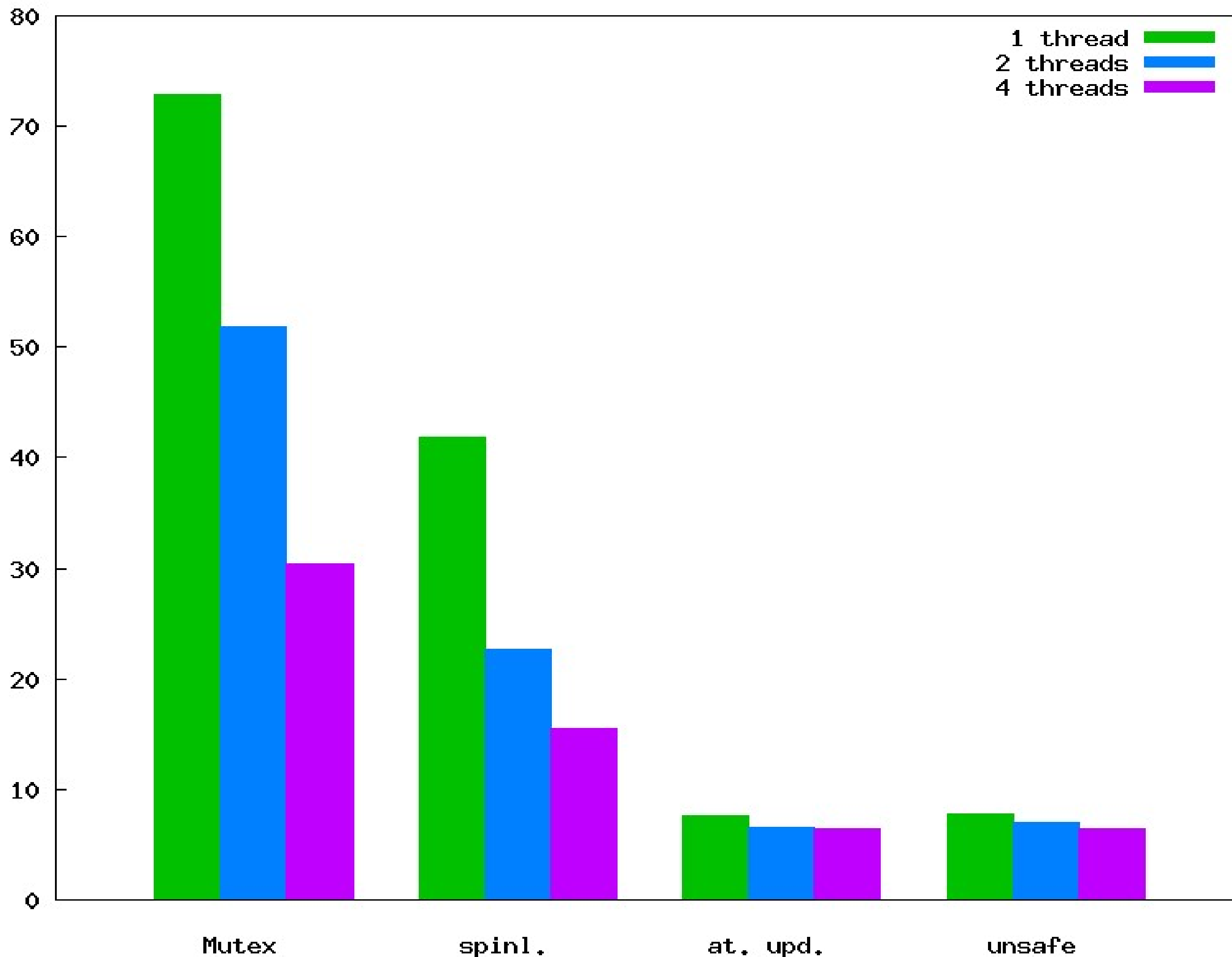


Itanium (gcc) running time 4x1GHz, bits



invent

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.