Nondeterminism is Unavoidable, but Data Races are Pure Evil

Hans-J. Boehm

HP Labs

Low-level nondeterminism is pervasive

- E.g.
  - Atomically incrementing a global counter is nondeterministic.
    - Which is unobservable if only read after threads are joined.
  - Memory allocation returns nondeterministic addresses.
    - Which usually doesn't matter.
    - But sometimes does.
  - Same for any other resource allocation.
- Harder to debug, but helps test coverage.
- These are races of a sort,
  - *but not data races*

Data Races

- Two memory accesses conflict if they
  - access the same memory location, e.g. variable
  - at least one access is a store
- A program has a data race if two *data* accesses
  - conflict, and
  - can occur simultaneously.
    - simultaneous = different threads and unordered by synchronization
- A program data-race-free (on a particular input) if no sequentially consistent execution results in a data race.

November 2010 CACM Technical Perspective

- **"Data races are evil with no exceptions."**

- Sarita Adve, technical perspective for Goldilocks and FastTrack papers on data race detection.

Data Races are Evil, because:

- They are errors or poorly defined in all mainstream multithreaded programming languages.
  - Treated like subscript errors in C11, C++11, pthreads, Ada83, OpenMP, ...
- May unpredictably break code in practice.
  - Now or when you recompile, upgrade your OS, or …
- Are easily avoidable in C11 & C++11.
- Don't improve scalability significantly.
  - Even if the code still works.

C++ 2011 / C1x

- [C++ 2011 1.10p21] The execution of a program contains a *data* race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Data races are errors!

(and you get atomic operations to avoid them)

Ada 83

- [ANSI-STD-1815A-1983, 9.11] For the actions performed by a program that uses shared variables, the following assumptions can always be made:
  - If between two synchronization points in a task, this task reads a shared variable whose type is a scalar or access type, then the variable is not updated by any other task at any time between these two points.
  - If between two synchronization points in a task, this task updates a shared variable whose task type is a scalar or access type, then the variable is neither read nor updated by any other task at any time between these two points.
- The execution of the program is erroneous if any of these assumptions is violated.

Data races are errors!

Posix Threads Specification

- [IEEE 1003.1-2008, Base Definitions 4.11; dates back to 1995] 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.

Data races are errors!

Java

Data races are not errors.

But we don't really know what they really mean.

(e.g. [Sevcik, Aspinall ECOOP 2008])

Absence of data races guarantees sane semantics

But does it really matter in practice?

Code breakage example

```
bool done; int x;
```

Thread 1:

```
x = 42;
done = true;
```

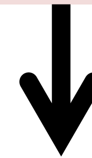
Thread 2:

```
while (!done) {}
assert(x == 42);
```

Code breakage example (2)

```
bool done; int x;
```

```
while (!done) {}
assert(x == 42);
```



Thread 1:

```
x = 42;
done = true;
```

Thread 2:

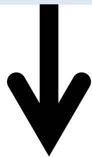
```
tmp = done;
while (!tmp) {}
assert(x == 42);
```

Compiler

Code breakage example (3)

```
bool done; int x;
```

```
x = 42;
done = true;
```



Thread 1:

```
done = true;
x = 42;
```

Thread 2:

```
while (!done) {}
assert(x == 42);
```

Compiler, ARM, POWER

Code breakage example (4)

```
bool done; int x;
```

Load of x precedes final load of done

Thread 1:

```
x = 42;
done = true;
```

Thread 2:

```
while (!done) {}
assert(x == 42);
```

ARM, POWER

Another code breakage example

```
{
  bool tmp = x;  // racy load
  if (tmp)
     launch_missile = true;
  …
  if (tmp && no_incoming)
     launch_missile = false;
}
```

Another code breakage example

```
{
  bool tmp = x;  // racy load
  if (tmp)
      launch_missile = true;
  …
  if (tmp && no_incoming)
      launch_missile = false;
}
```

Other “misoptimizations”

- Reading half a pointer.
- Reading uninitialized method table pointer.
- See HotPar ‘11 paper.
  - Even redundant writes can fail.

But if it works, does it help performance?

- C11 and C++11 have two main ways to avoid data races:
  1. Locks
  2. `atomic` objects
     - Indivisible operations.
     - Treated as synchronization; don't create data races.
     - Warns compiler of the race.

Code breakage example fixed (1)

```
atomic<bool> done; int x;
```

Thread 1:

```
x = 42;
done = true;
```

Thread 2:

```
while (!done) {}
assert(x == 42);
```

Code breakage example fixed (2)

```
atomic<bool> done; int x;
```

Thread 1:

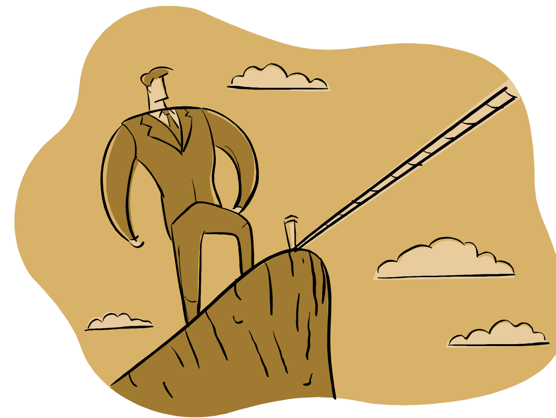
```
x = 42;
done.store(true,
  memory_order_release);
```

Thread 2:

```
while (!done.load(
    memory_order_acquire) {}
assert(x == 42);
```

Complex, and dangerous

but much simpler and safer than data races



So how much does synchronization cost?

- Architecture dependent.
- On X86:
  - Atomic stores with any ordering constraint except default `memory_order_seq_cst` impose only mild compiler constraints.
    - If data races work, they can probably generate the same code.
  - Atomic loads impose only mild compiler constraints.

So how much does “non-free” synchronization cost?

- Locking read-only critical sections can be expensive.
  - Adds cache contention where there was none.
  - And contention is the real scalability issue.
  - *But if races “work”, then atomics work better.*
  - And on X86 atomic reads are “free”.
  - And there are cheaper ways to handle read-only “critical sections” (seqlocks, RCU). See my MSPC ‘12 paper.
- Here we focus on critical sections that write.

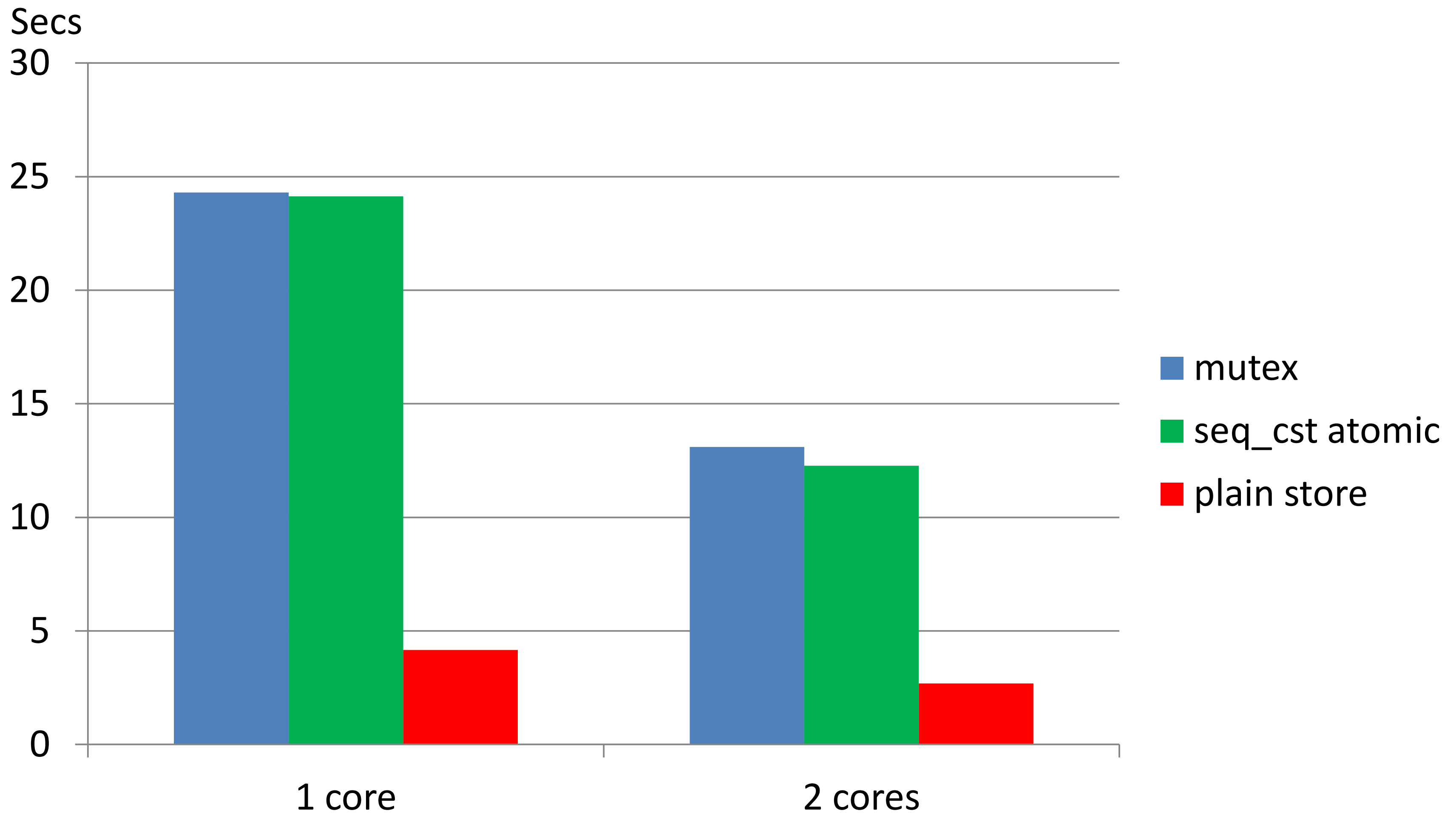
Simple benchmark

- Sieve of Eratosthenes example from PLDI 05 paper, modified to write unconditionally:

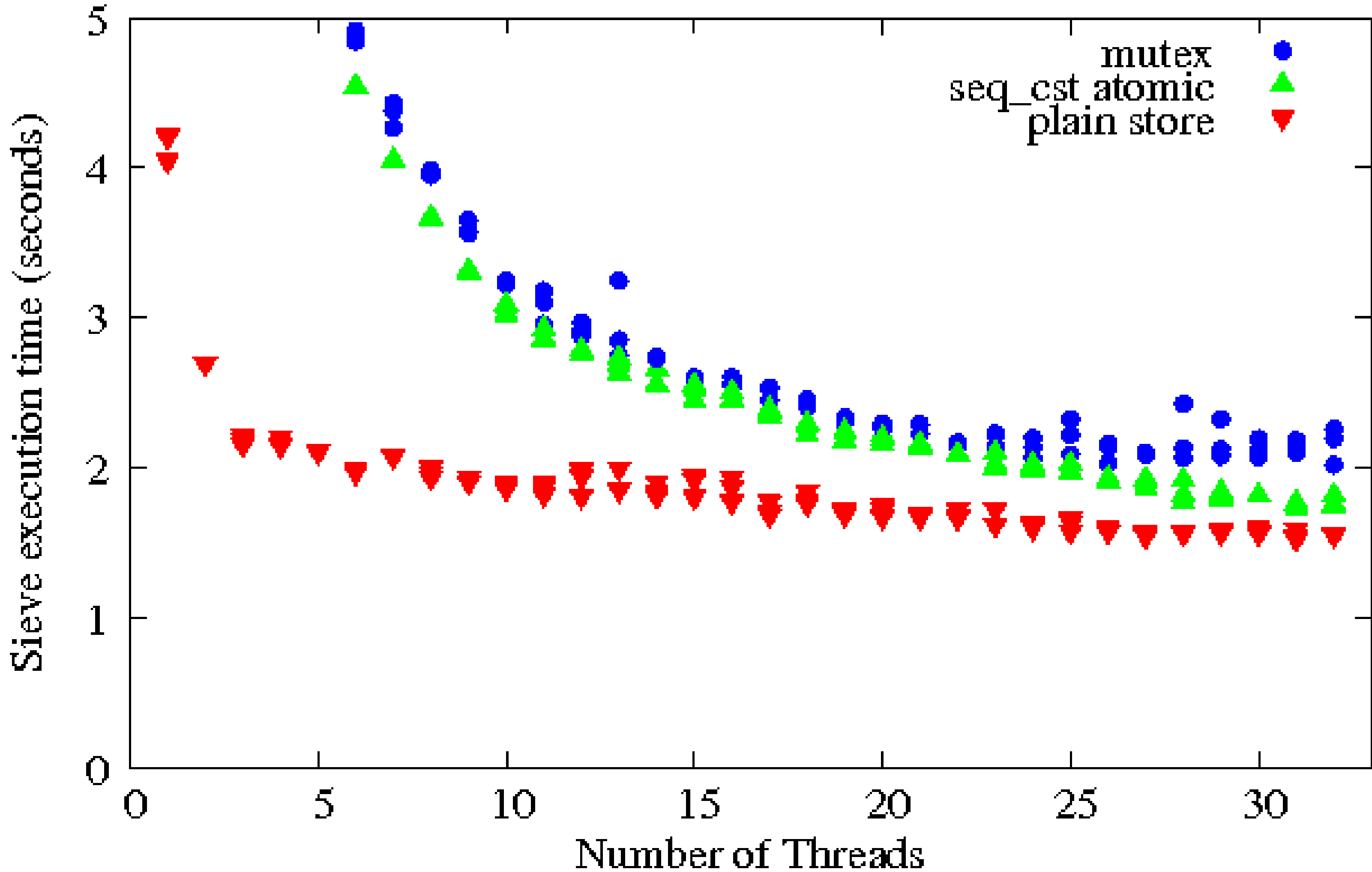
```
for (my_prime = start; my_prime < SQRT; ++my_prime)
  if (!get(my_prime)) {
    for (multiple = my_prime;
         multiple < MAX; multiple += my_prime)
      set(multiple);
  }
```

- Ran on 8 processors x 4 core x86 AMD machine.
- C++11 atomic-equivalents hand-generated.
- *Lock individual accesses.*

Time for 10^8 primes, 1 and 2 cores



Time for 10^8 primes, more cores



Why?

- Conjecture: Memory-bandwidth limited.
- `memory_order_seq_cst` adds `mfence` instruction to store.
  - Approximate effect: Flush store buffer.
    - Local; no added memory traffic.
- Locking adds fences + some memory traffic.
  - Locks are much smaller than bit array. (hashed)
  - Tuned to minimize memory bandwidth and contention.
  - Also mostly local, parallelizable overhead.

Conclusion

Questions?