Concurrency as a Library

- Many common languages are designed without native concurrency support.
- Consequently, many of these languages have libraries that provide locking primitives.
- This *mostly* works, but there are some unpleasant surprises that are non-obvious.
Fundamental Problems

- Many of the problems with multiprocessors lie at the hardware or compiler levels, and thus cannot be addressed with libraries.
- This requires a good understanding of underlying implementation to do this correctly.
Overview of sequential consistency

● View concurrent execution as an “interleaving of the steps from the threads.”

\[
\begin{align*}
x &= 1; & r1 &= y; \\
y &= 1; & r2 &= x; \\
\end{align*}
\]

● One or both of r1 and r2 must have the value 1.
The real world

- This model is inefficient and disallows many types of optimizations.
- Nearly all modern architectures (like x86) allow the hardware to reorder memory operations.
- Compilers will generally reorder memory loads and stores for efficiency.
The Pthreads way

- Pthreads uses functions that guaranteed to “synchronize memory”. These include hardware instructions to prevent memory reordering.
- `pthread_mutex_lock()` and others
- The compiler treats them as black boxes which may touch global memory, so it won’t move memory across a function call.
Surprises

- This approach is not precise enough.
- Programs may fail intermittently, or when using a new compiler or hardware.
- Thread ordering is often non-deterministic, making this trickier to debug.
- In some cases, it excludes efficient algorithms
Consider this example

\begin{verbatim}
if (x == 1) ++y;
if (y == 1) ++x;
\end{verbatim}
Consider this example

Becomes

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

Outcome of the later is not consistent.
Adjacent data

**struct** { int a:17; int b:15 } x;
**becomes**

{ 
    tmp = x; // Read both fields into 
    // 32-bit variable.
    tmp &= ~0x1fffff; // Mask off old a.
    tmp |= 42;
    x = tmp; // Overwrite all of x.
}
struct { char a; char b; char c; char d; char e; char f; char g; char h; } x;

Let’s say you wish to assign each variable independently:

x.b = 'b'; x.c = 'c'; x.d = 'd'; x.e = 'e'; x.f = 'f'; x.g = 'g'; x.h = 'h';
Adjacent data

Instead of assigning each variable independently, the compiler may compress that to.

\[ x = 'hgfedcb\0' | x.a; \]

Pthreads allows this, but this may lead to non-portable code. Why?
Register promotion

- It is unsafe to promote variables in a critical section to places outside the critical section.
Register Promotion Example

for (...) {
    ...
    if (mt) pthread_mutex_lock(...);
    x = ... x ...
    if (mt) pthread_mutex_unlock(...);
}

This gets transformed to
Register Promotion Example

\[
\begin{align*}
&\text{r} = \text{x;} \\
&\text{for} \ (\ldots) \ \{ \\
&\quad \ldots \\
&\quad \text{if} \ (\text{mt}) \ \{ \\
&\quad \quad \text{x} = \text{r;} \ \text{pthread\_mutex\_lock}(\ldots); \ \text{r} = \text{x;} \\
&\quad \} \\
&\quad \text{r} = \ldots \ \text{r} \ldots \\
&\quad \text{if} \ (\text{mt}) \ \{ \\
&\quad \quad \text{x} = \text{r;} \ \text{pthread\_mutex\_unlock}(\ldots); \ \text{r} = \text{x;} \\
&\quad \} \\
&\} \\
&\text{x} = \text{r;} 
\end{align*}
\]
Performance Issues

● Pthreads assumes that memory operation order is irrelevant unless explicitly specified by the coder.
● This strategy precludes lock-free and wait-free strategies.
● The performance of these strategies is lost when reordering is restricted.
Figure 1: Sieve execution time for byte array (secs)