Software and the Concurrency Revolution

Herb Sutter
Software Architect
Microsoft Developer Division

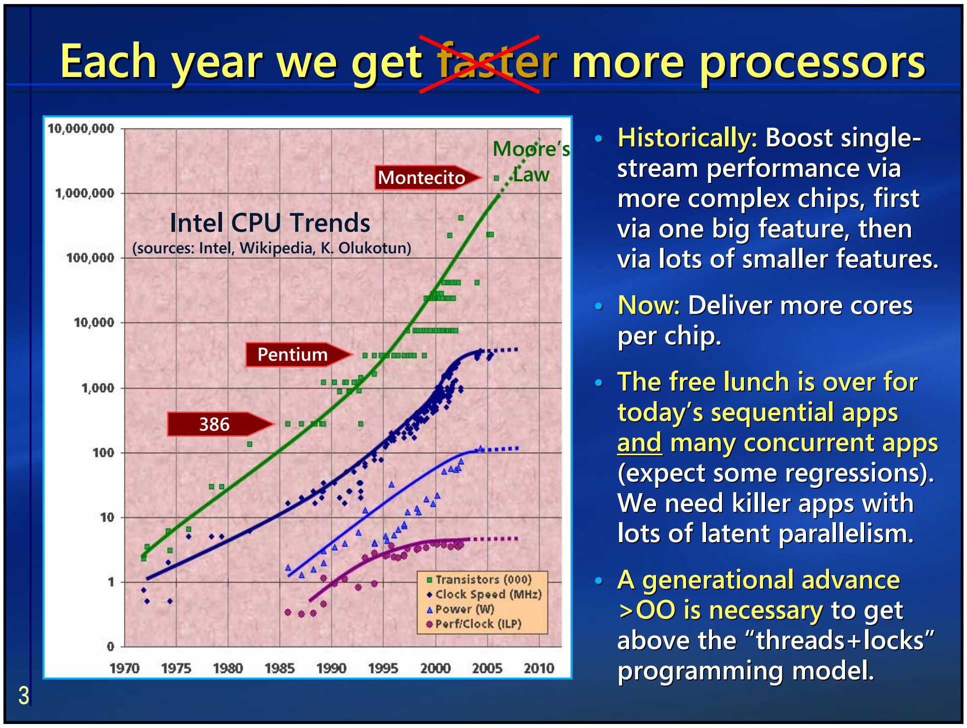
Concurrency

Truths

Consequences

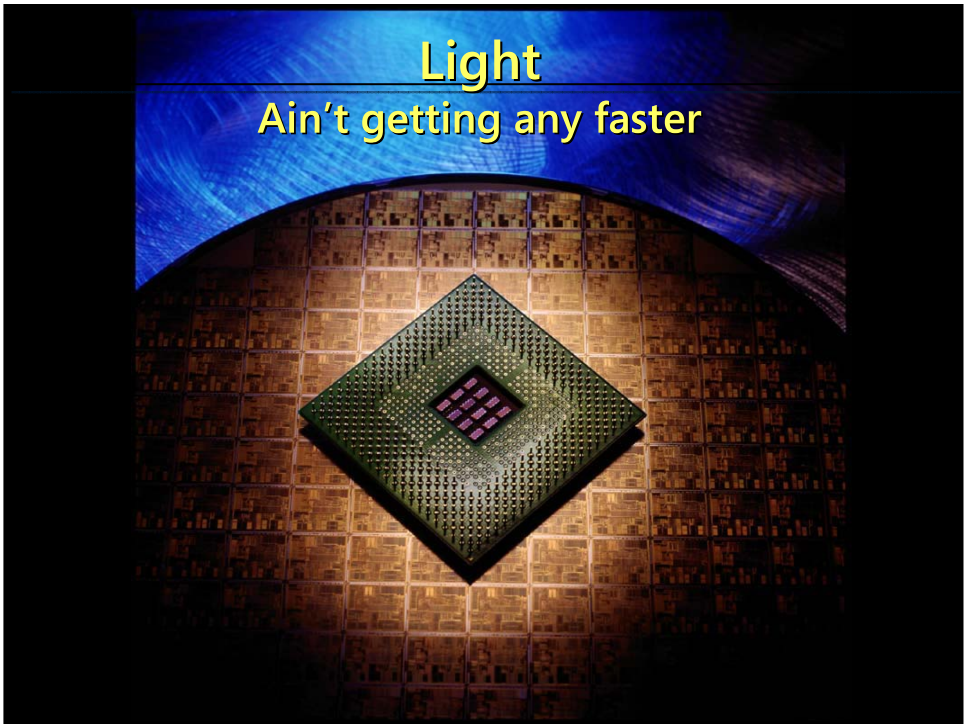
Futures

von Neumann



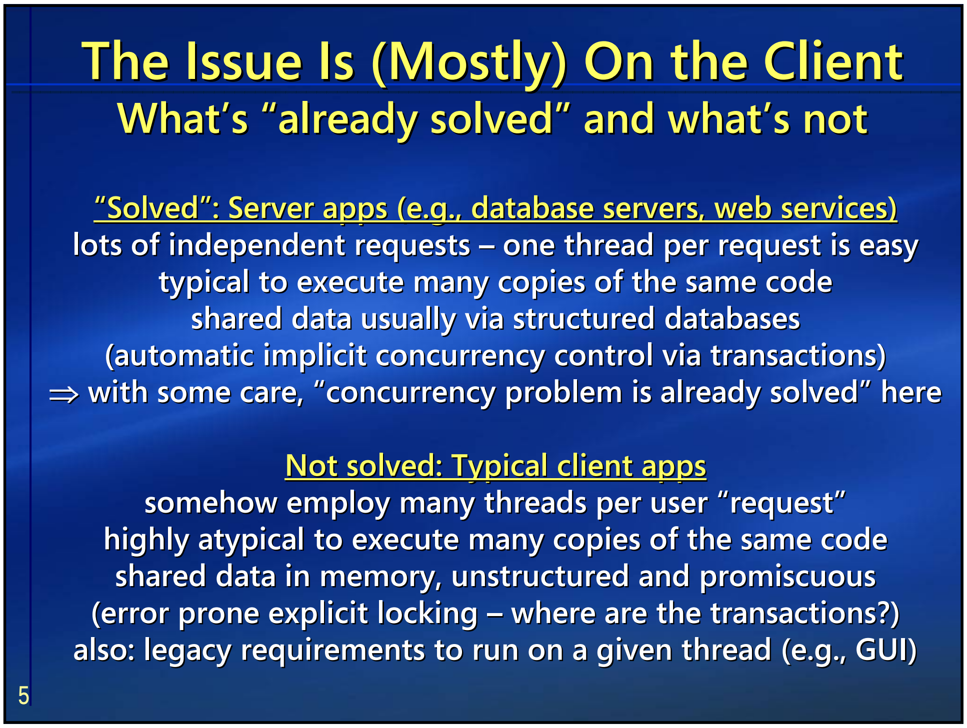
Each year we get ~~faster~~ more processors

- Historically: Boost single-stream performance via more complex chips, first via one big feature, then via lots of smaller features.
- Now: Deliver more cores per chip.
- The free lunch is over for today's sequential apps and many concurrent apps (expect some regressions). We need killer apps with lots of latent parallelism.
- A generational advance >OO is necessary to get above the "threads+locks" programming model.



Light

Ain't getting any faster

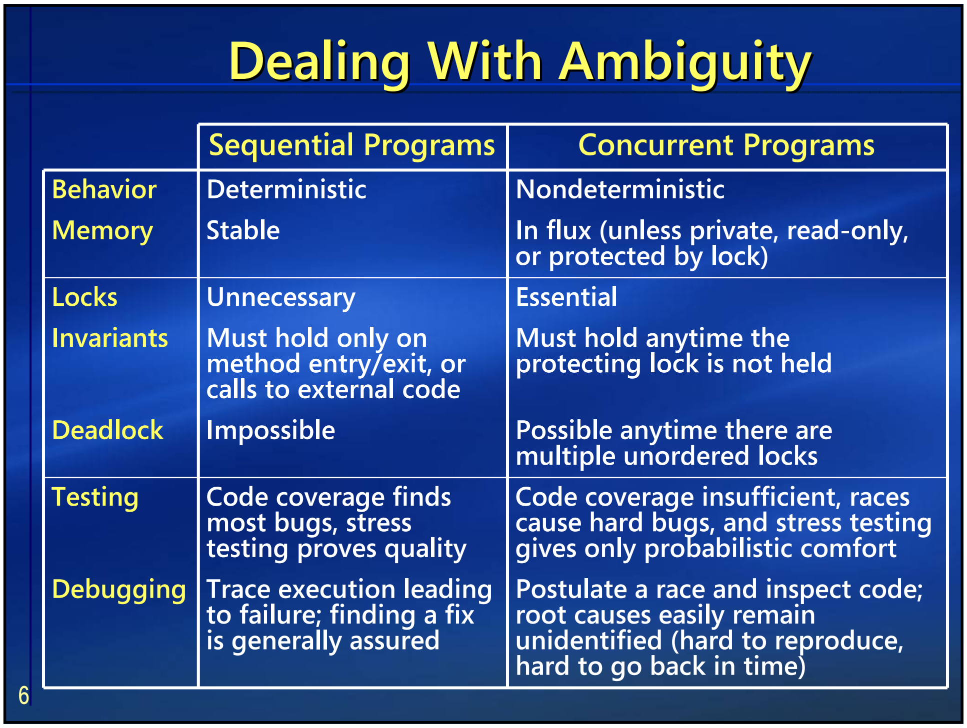


The Issue Is (Mostly) On the Client

What's "already solved" and what's not

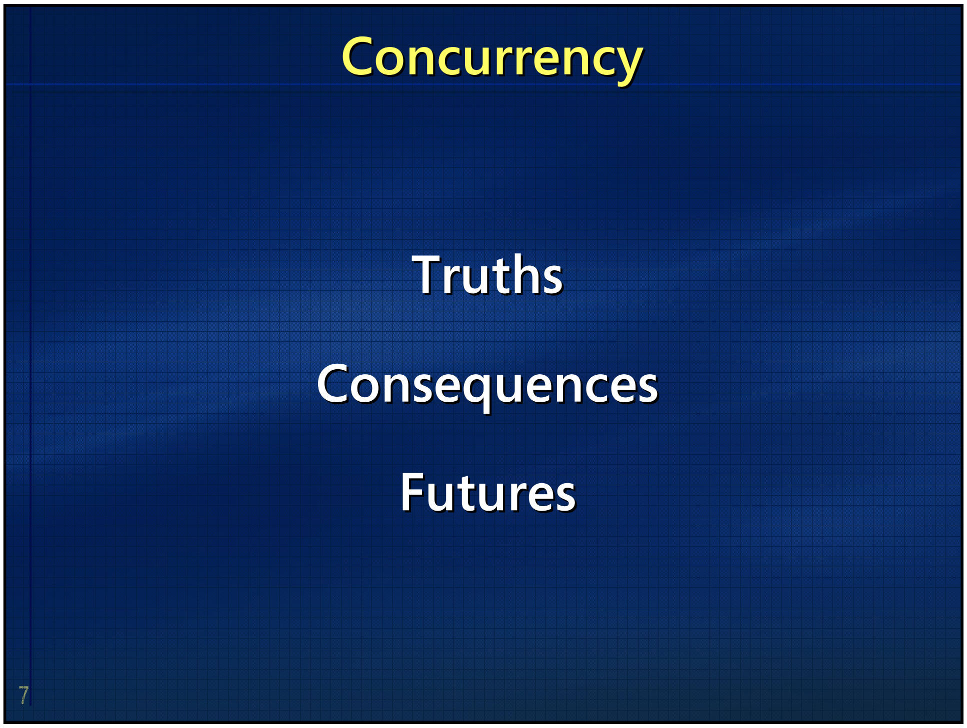
"Solved": Server apps (e.g., database servers, web services)
lots of independent requests – one thread per request is easy
typical to execute many copies of the same code
shared data usually via structured databases
(automatic implicit concurrency control via transactions)
⇒ with some care, "concurrency problem is already solved" here

Not solved: Typical client apps
somehow employ many threads per user "request"
highly atypical to execute many copies of the same code
shared data in memory, unstructured and promiscuous
(error prone explicit locking – where are the transactions?)
also: legacy requirements to run on a given thread (e.g., GUI)



Dealing With Ambiguity

	Sequential Programs	Concurrent Programs
Behavior	Deterministic	Nondeterministic
Memory	Stable	In flux (unless private, read-only, or protected by lock)
Locks	Unnecessary	Essential
Invariants	Must hold only on method entry/exit, or calls to external code	Must hold anytime the protecting lock is not held
Deadlock	Impossible	Possible anytime there are multiple unordered locks
Testing	Code coverage finds most bugs, stress testing proves quality	Code coverage insufficient, races cause hard bugs, and stress testing gives only probabilistic comfort
Debugging	Trace execution leading to failure; finding a fix is generally assured	Postulate a race and inspect code; root causes easily remain unidentified (hard to reproduce, hard to go back in time)

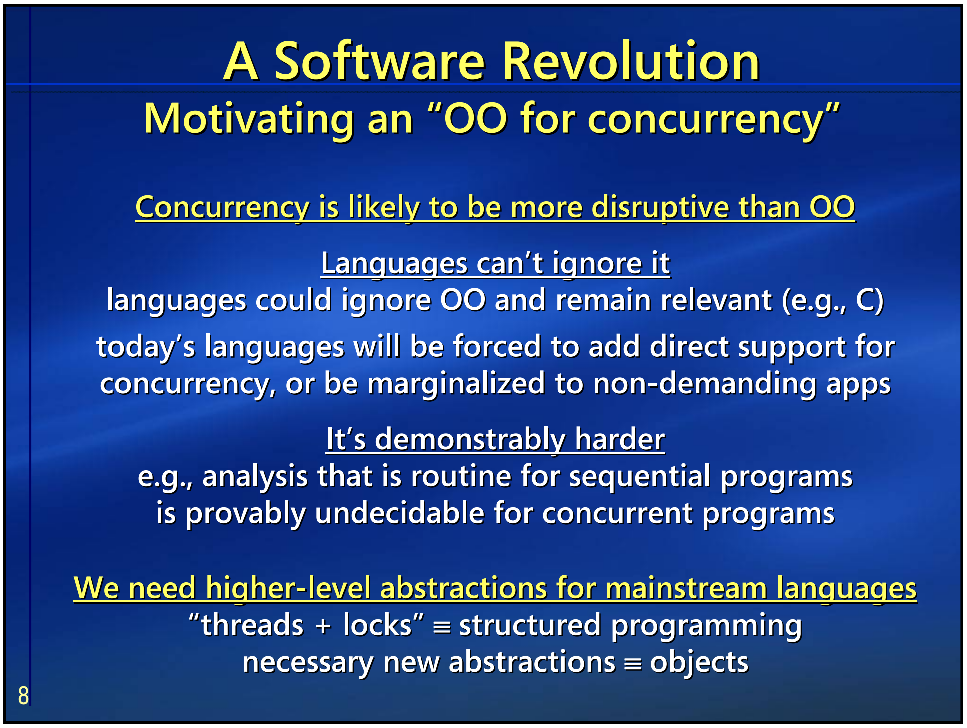


Concurrency

Truths

Consequences

Futures



A Software Revolution

Motivating an “OO for concurrency”

Concurrency is likely to be more disruptive than OO

Languages can’t ignore it

languages could ignore OO and remain relevant (e.g., C)

today’s languages will be forced to add direct support for concurrency, or be marginalized to non-demanding apps

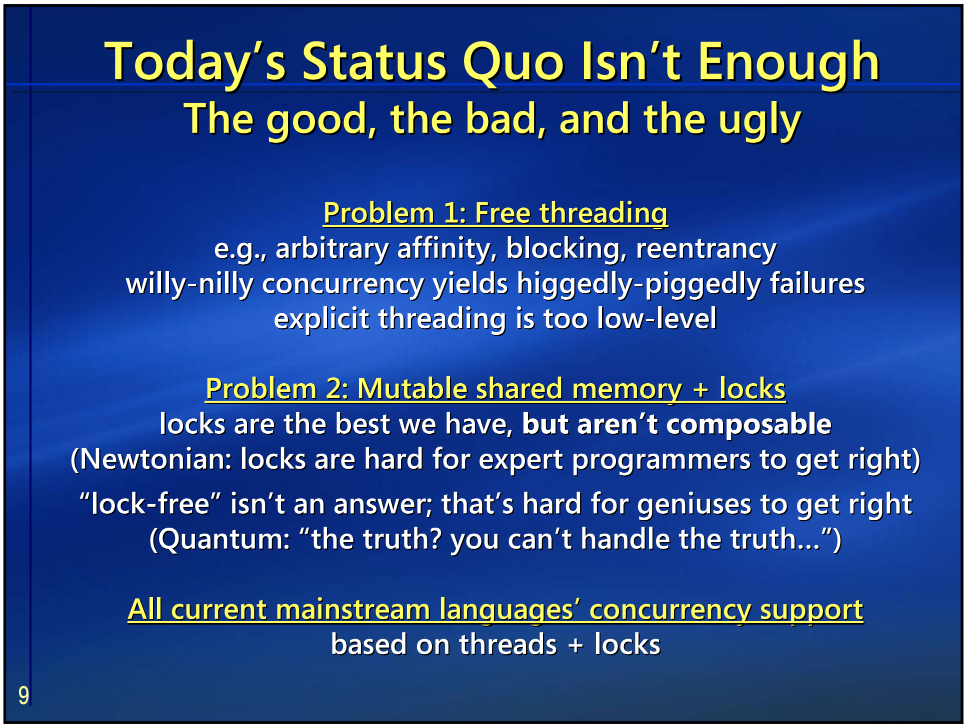
It’s demonstrably harder

e.g., analysis that is routine for sequential programs is provably undecidable for concurrent programs

We need higher-level abstractions for mainstream languages

“threads + locks” ≡ structured programming

necessary new abstractions ≡ objects



Today’s Status Quo Isn’t Enough

The good, the bad, and the ugly

Problem 1: Free threading

e.g., arbitrary affinity, blocking, reentrancy

willy-nilly concurrency yields higgedly-piggedly failures

explicit threading is too low-level

Problem 2: Mutable shared memory + locks

locks are the best we have, **but aren’t composable**

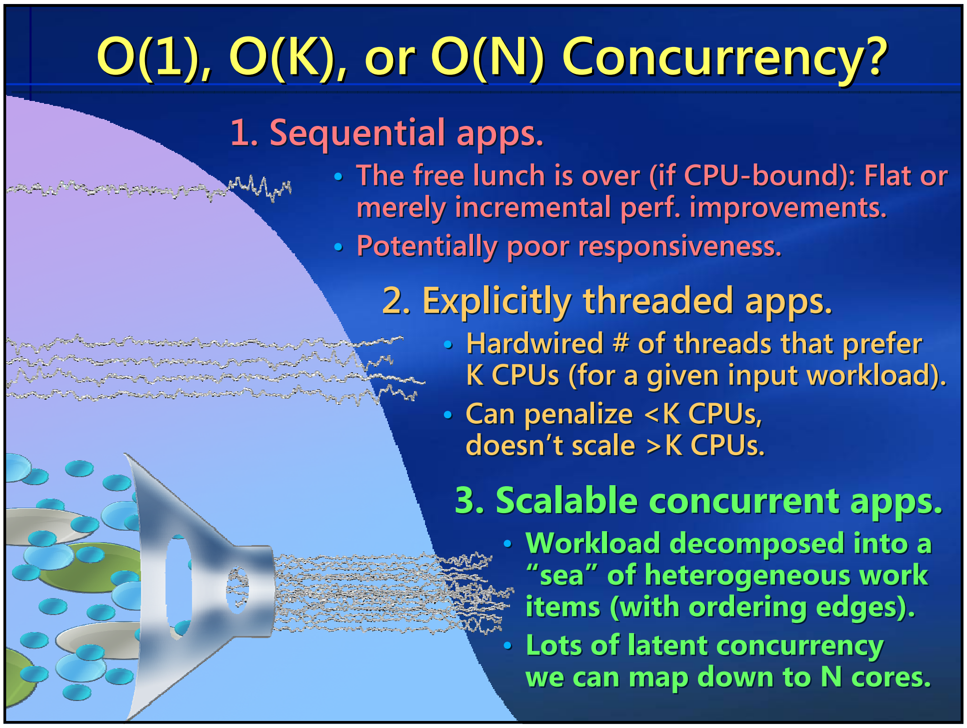
(Newtonian: locks are hard for expert programmers to get right)

“lock-free” isn’t an answer; that’s hard for geniuses to get right

(Quantum: “the truth? you can’t handle the truth…”)

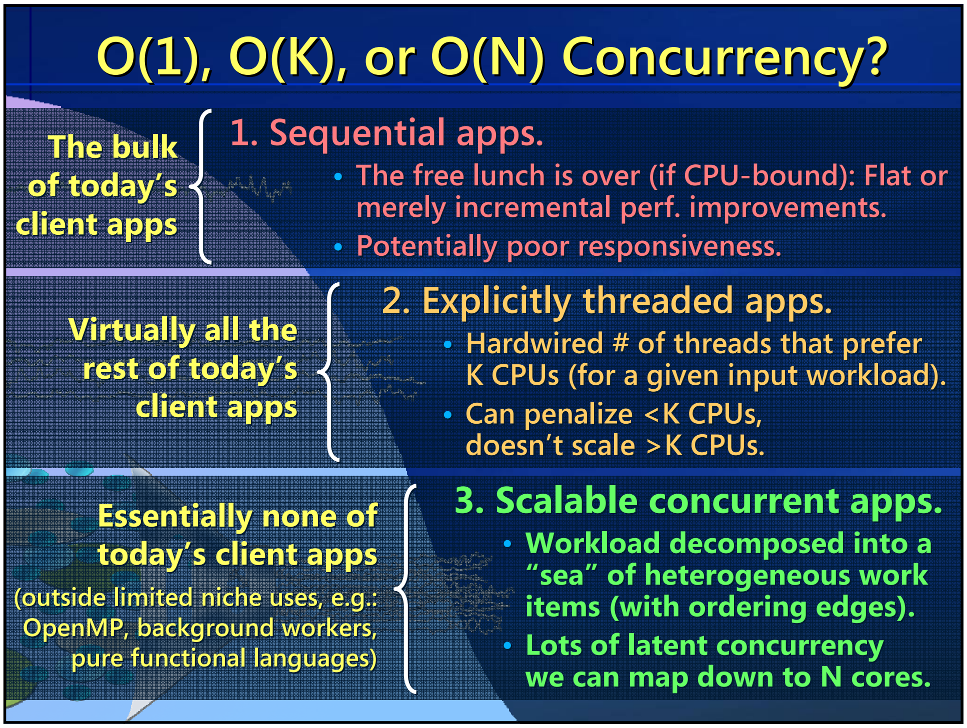
All current mainstream languages’ concurrency support

based on threads + locks



O(1), O(K), or O(N) Concurrency?

1. Sequential apps.
   - The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
   - Potentially poor responsiveness.
2. Explicitly threaded apps.
   - Hardwired # of threads that prefer K CPUs (for a given input workload).
   - Can penalize <K CPUs, doesn't scale >K CPUs.
3. Scalable concurrent apps.
   - Workload decomposed into a "sea" of heterogeneous work items (with ordering edges).
   - Lots of latent concurrency we can map down to N cores.



O(1), O(K), or O(N) Concurrency?

The bulk of today's client apps

1. Sequential apps.
   - The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
   - Potentially poor responsiveness.

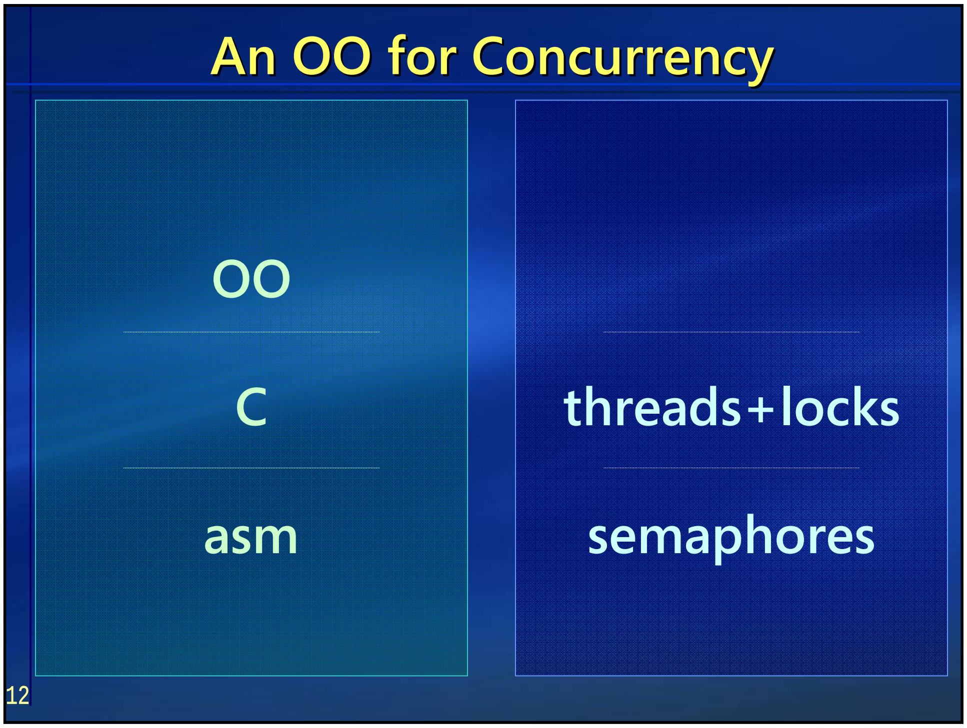
Virtually all the rest of today's client apps

2. Explicitly threaded apps.
   - Hardwired # of threads that prefer K CPUs (for a given input workload).
   - Can penalize <K CPUs, doesn't scale >K CPUs.

Essentially none of today's client apps

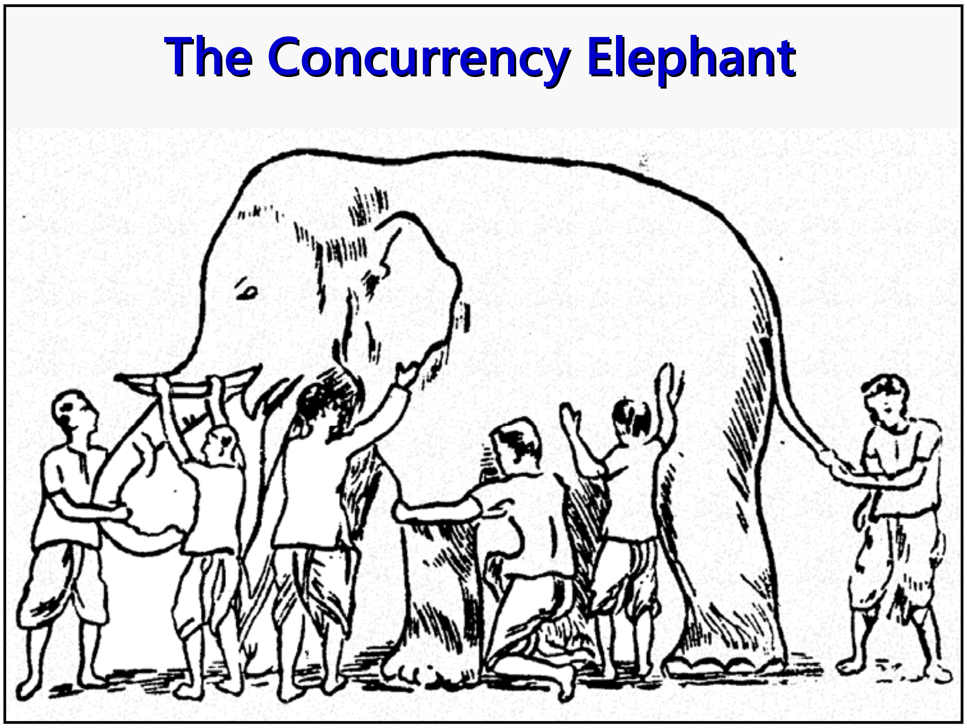
(outside limited niche uses, e.g.: OpenMP, background workers, pure functional languages)

3. Scalable concurrent apps.
   - Workload decomposed into a "sea" of heterogeneous work items (with ordering edges).
   - Lots of latent concurrency we can map down to N cores.

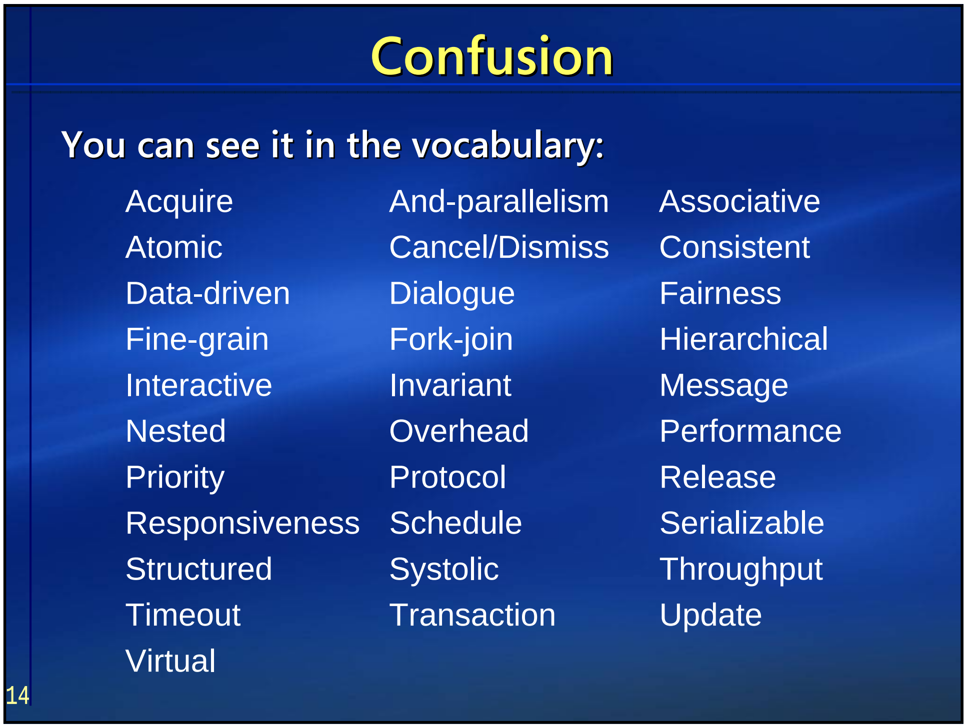


An OO for Concurrency

OO	
C	threads+locks
asm	semaphores



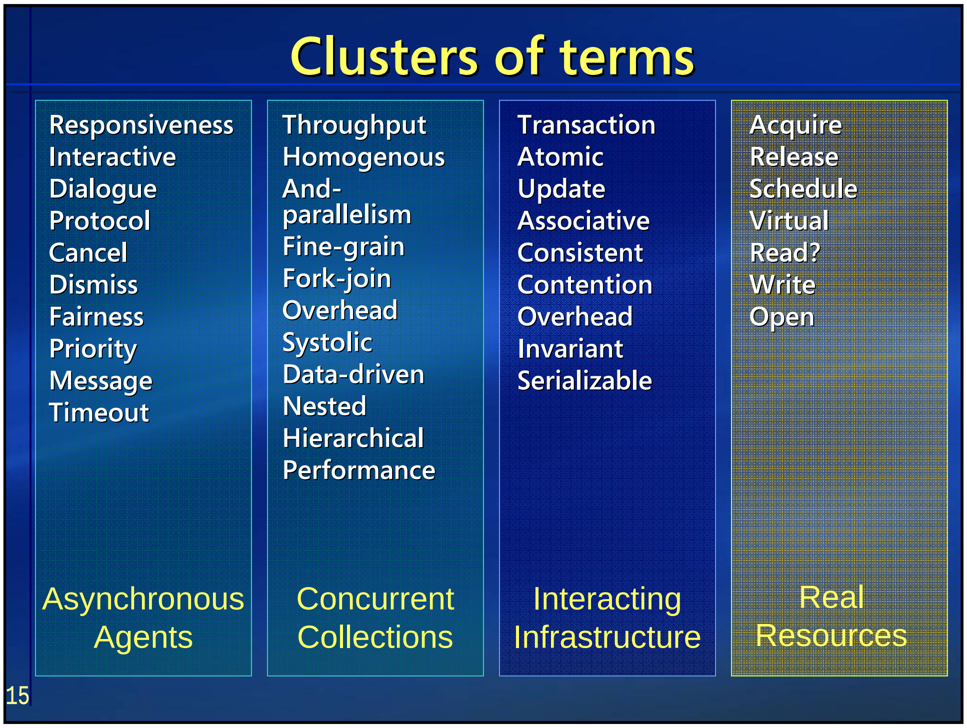
The Concurrency Elephant



Confusion

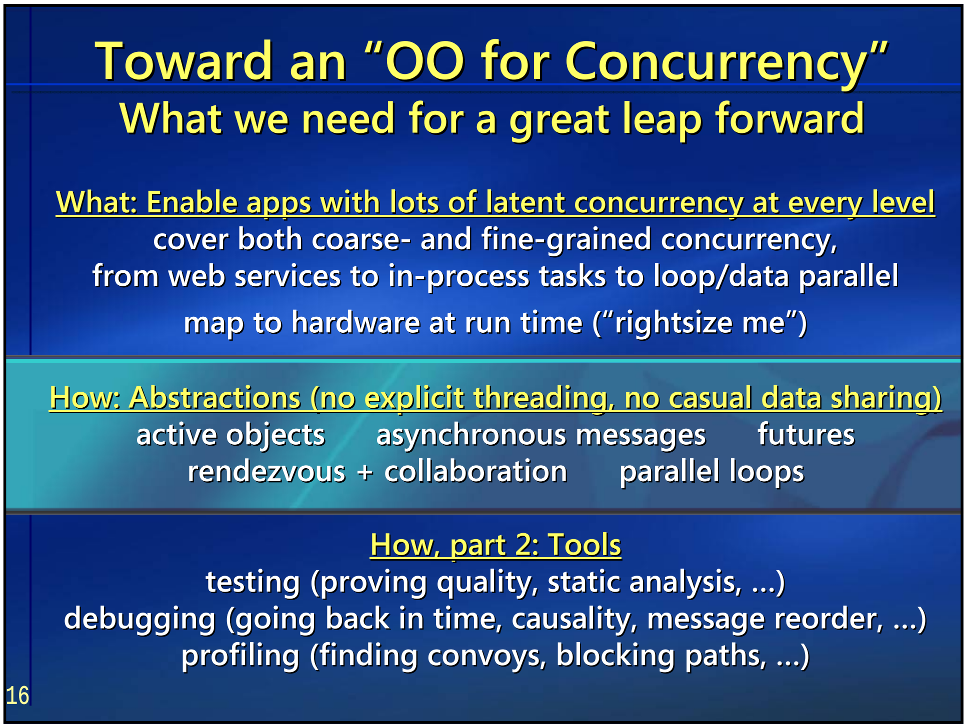
You can see it in the vocabulary:

- Acquire
- And-parallelism
- Associative
- Atomic
- Cancel/Dismiss
- Consistent
- Data-driven
- Dialogue
- Fairness
- Fine-grain
- Fork-join
- Hierarchical
- Interactive
- Invariant
- Message
- Nested
- Overhead
- Performance
- Priority
- Protocol
- Release
- Responsiveness
- Schedule
- Serializable
- Structured
- Systolic
- Throughput
- Timeout
- Transaction
- Update
- Virtual



Clusters of terms

Asynchronous Agents	Concurrent Collections	Interacting Infrastructure	Real Resources
Responsiveness	Throughput	Transaction	Acquire
Interactive	Homogenous	Atomic	Release
Dialogue	And-parallelism	Update	Schedule
Protocol	Fine-grain	Associative	Virtual
Cancel	Fork-join	Consistent	Read?
Dismiss	Overhead	Contention	Write
Fairness	Systolic	Overhead	Open
Priority	Data-driven	Invariant	
Message	Nested	Serializable	
Timeout	Hierarchical		
	Performance		



Toward an "OO for Concurrency"

What we need for a great leap forward

What: Enable apps with lots of latent concurrency at every level

cover both coarse- and fine-grained concurrency,
from web services to in-process tasks to loop/data parallel

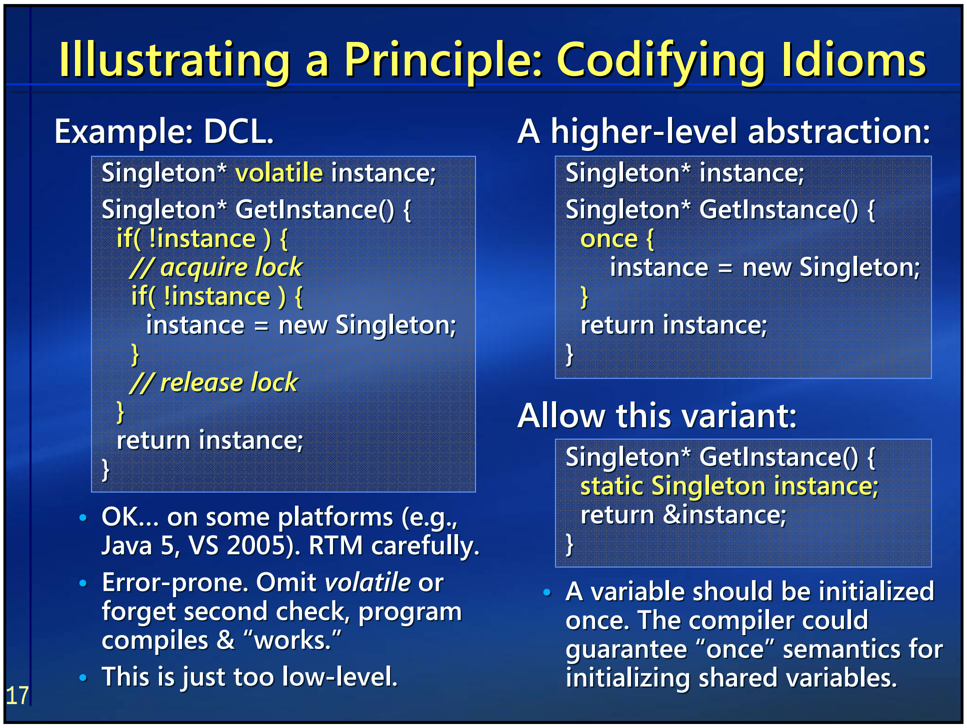
map to hardware at run time ("rightsize me")

How: Abstractions (no explicit threading, no casual data sharing)

active objects asynchronous messages futures
rendezvous + collaboration parallel loops

How, part 2: Tools

testing (proving quality, static analysis, ...)
debugging (going back in time, causality, message reorder, ...)
profiling (finding convoys, blocking paths, ...)



Illustrating a Principle: Codifying Idioms

Example: DCL.

```
Singleton* volatile instance;
Singleton* GetInstance() {
  if( !instance ) {
    // acquire lock
    if( !instance ) {
      instance = new Singleton;
    }
    // release lock
  }
  return instance;
}
```

- OK... on some platforms (e.g., Java 5, VS 2005). RTM carefully.
- Error-prone. Omit *volatile* or forget second check, program compiles & "works."
- This is just too low-level.

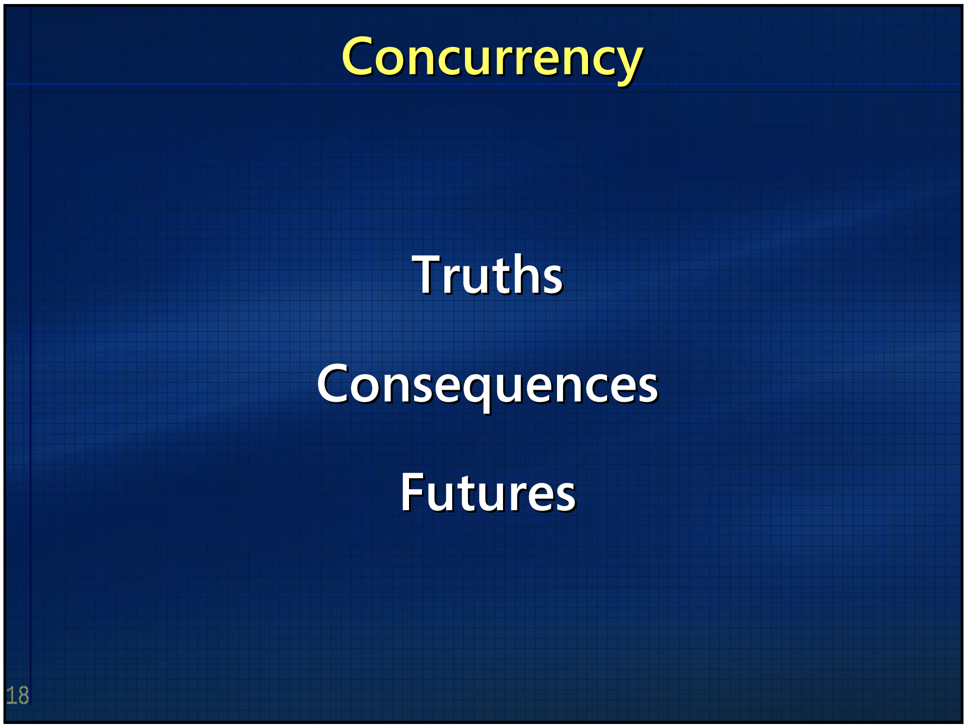
A higher-level abstraction:

```
Singleton* instance;
Singleton* GetInstance() {
  once {
      instance = new Singleton;
  }
  return instance;
}
```

Allow this variant:

```
Singleton* GetInstance() {
  static Singleton instance;
  return &instance;
}
```

- A variable should be initialized once. The compiler could guarantee "once" semantics for initializing shared variables.

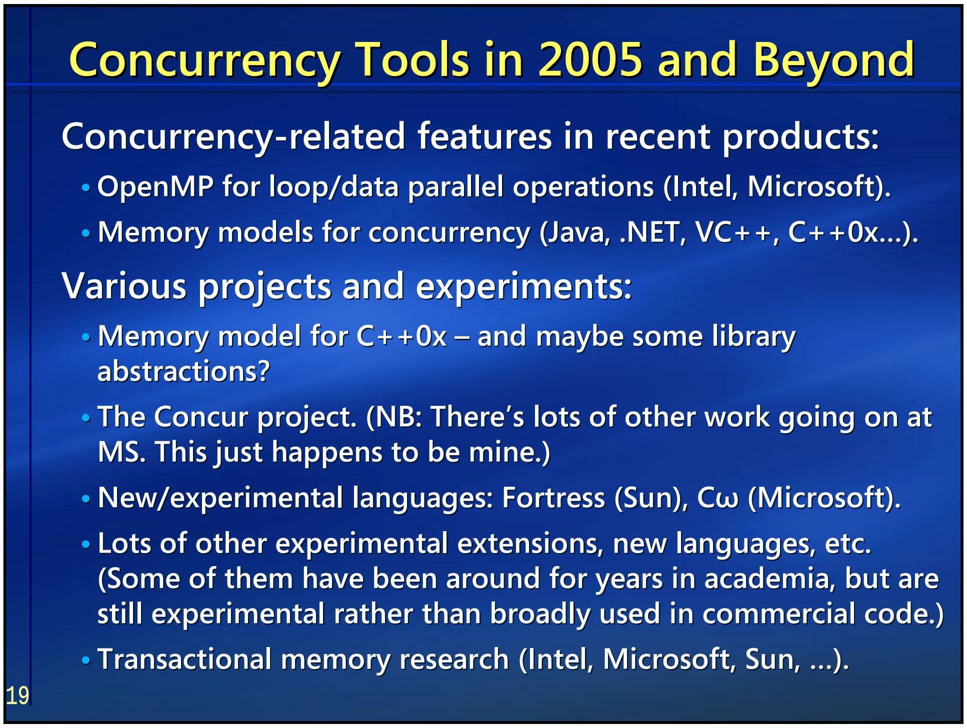


Concurrency

Truths

Consequences

Futures



Concurrency Tools in 2005 and Beyond

Concurrency-related features in recent products:

- OpenMP for loop/data parallel operations (Intel, Microsoft).
- Memory models for concurrency (Java, .NET, VC++, C++0x…).

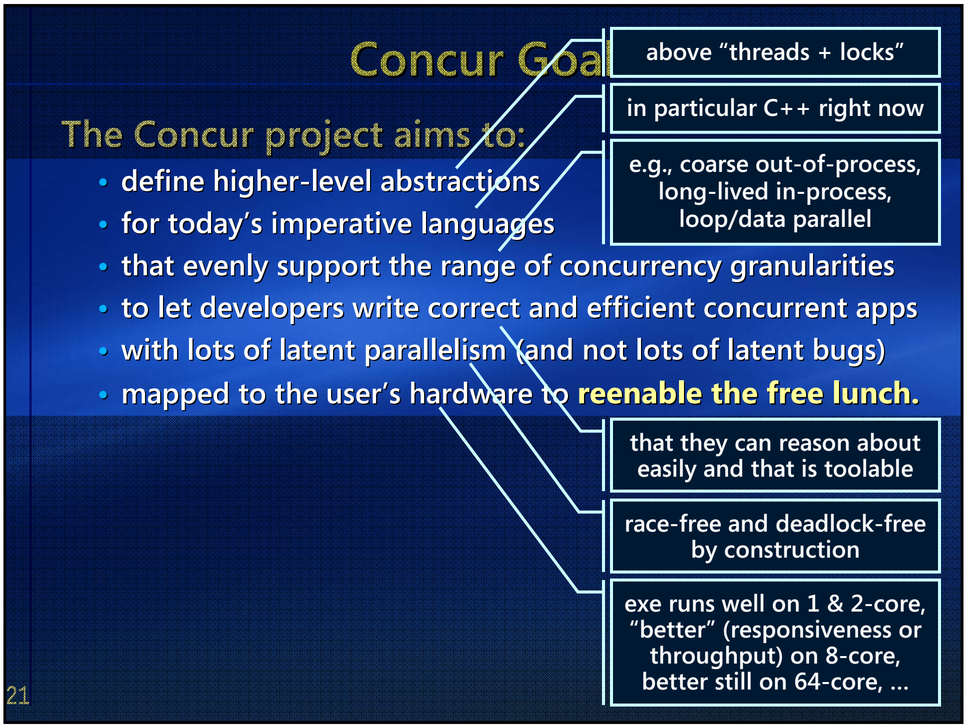
Various projects and experiments:

- Memory model for C++0x – and maybe some library abstractions?
- The Concur project. (NB: There's lots of other work going on at MS. This just happens to be mine.)
- New/experimental languages: Fortress (Sun), Cω (Microsoft).
- Lots of other experimental extensions, new languages, etc. (Some of them have been around for years in academia, but are still experimental rather than broadly used in commercial code.)
- Transactional memory research (Intel, Microsoft, Sun, …).

Concur Goals

The Concur project aims to:

- define higher-level abstractions
- for today's imperative languages
- that evenly support the range of concurrency granularities
- to let developers write correct and efficient concurrent apps
- with lots of latent parallelism (and not lots of latent bugs)
- mapped to the user's hardware to **reenable the free lunch.**



Concur Goals

The Concur project aims to:

- define higher-level abstractions
- for today's imperative languages
- that evenly support the range of concurrency granularities
- to let developers write correct and efficient concurrent apps
- with lots of latent parallelism (and not lots of latent bugs)
- mapped to the user's hardware to **reenable the free lunch.**

above "threads + locks"

in particular C++ right now

e.g., coarse out-of-process, long-lived in-process, loop/data parallel

that they can reason about easily and that is toolable

race-free and deadlock-free by construction

exe runs well on 1 & 2-core, "better" (responsiveness or throughput) on 8-core, better still on 64-core, …

Concur Goals

The Concur project aims to:

- define higher-level abstractions
- for today's imperative languages
- that evenly support the range of concurrency granularities
- to let developers write correct and efficient concurrent apps
- with lots of latent parallelism (and not lots of latent bugs)
- mapped to the user's hardware to **reenable the free lunch.**

Eliminate/reduce "threads+locks":

- Blocking and reentrancy: Never silently or by default, always explicit and controlled by higher-level abstractions.
- Isolation: On active object boundaries + ownership semantics (e.g., transfer/lending). Reduce mutable sharing & locking.
- Locks: Declarative support for associating data with locks, expressing lock levels, etc. Support static/dynamic analysis.

50,000' View: Producing the Sea

Active objects/blocks.

```
active C c;

c.f();                  // these calls are nonblocking; each method
c.g();                  // call automatically enqueues message for c

...                     // this code can execute in parallel with f & g
```

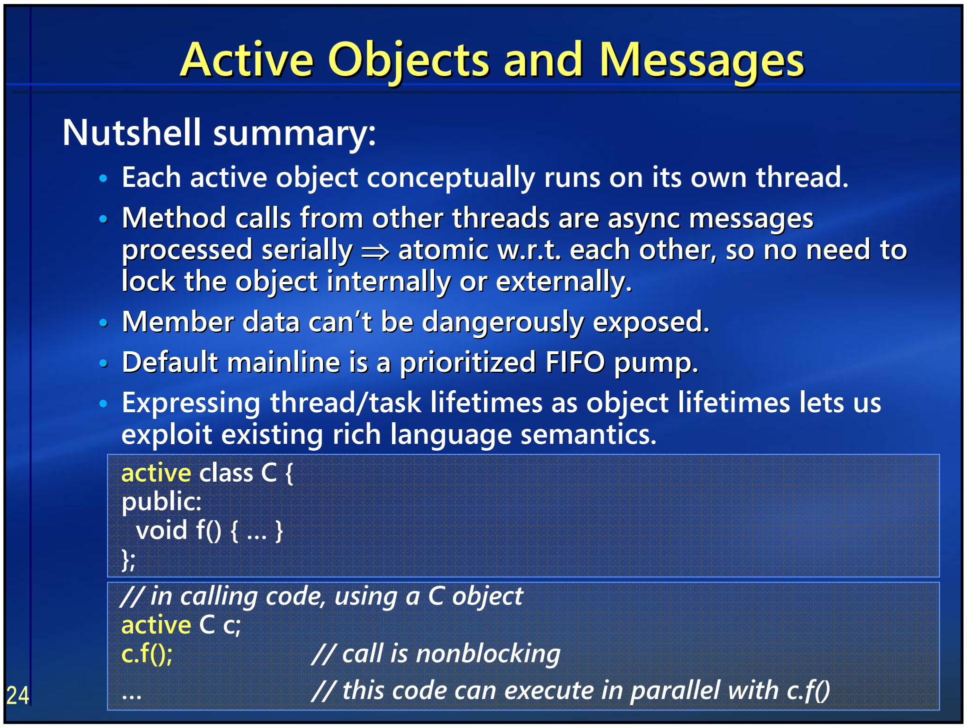
```
x = active { /*...*/ return foo(10); }; // do some work asynchronously
y = active { a->b( c ) };               // evaluate expr asynchronously
```

```
z = x.wait() * y.wait();                // express join points via futures
```

Parallel algorithms (sketch, under development).

```
for_each( c.depth_first(), f );                // sequential
for_each( c.depth_first(), f, parallel );      // fully parallel
for_each( c.depth_first(), f, ordered );       // ordered parallel
```

Gaining/losing concurrency is explicit: active and wait.



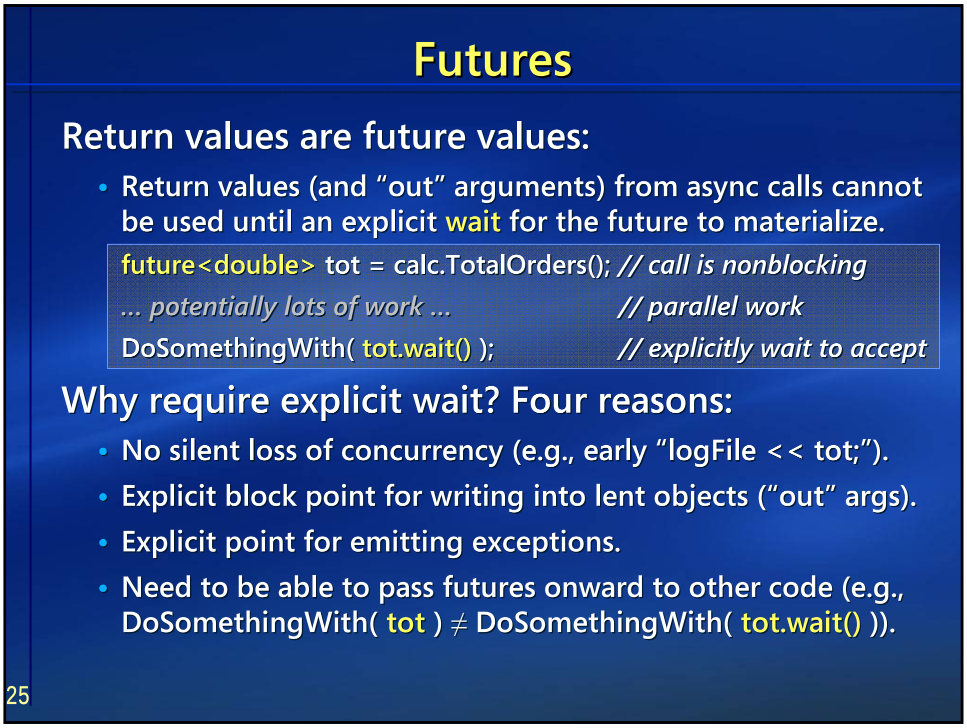
Active Objects and Messages

Nutshell summary:

- Each active object conceptually runs on its own thread.
- Method calls from other threads are async messages processed serially ⇒ atomic w.r.t. each other, so no need to lock the object internally or externally.
- Member data can't be dangerously exposed.
- Default mainline is a prioritized FIFO pump.
- Expressing thread/task lifetimes as object lifetimes lets us exploit existing rich language semantics.

```
active class C {
public:
  void f() { … }
};
```

```
// in calling code, using a C object
active C c;
c.f();             // call is nonblocking
…                  // this code can execute in parallel with c.f()
```



Futures

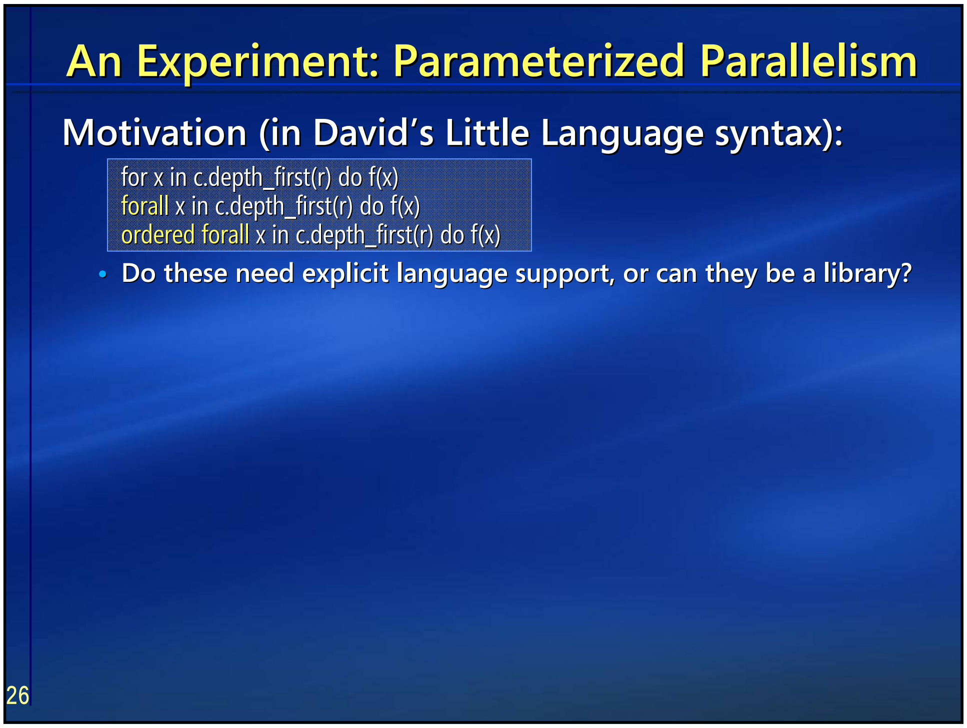
Return values are future values:

- Return values (and "out" arguments) from async calls cannot be used until an explicit wait for the future to materialize.

```
future<double> tot = calc.TotalOrders(); // call is nonblocking
… potentially lots of work …             // parallel work
DoSomethingWith( tot.wait() );           // explicitly wait to accept
```

Why require explicit wait? Four reasons:

- No silent loss of concurrency (e.g., early "logFile << tot;").
- Explicit block point for writing into lent objects ("out" args).
- Explicit point for emitting exceptions.
- Need to be able to pass futures onward to other code (e.g., DoSomethingWith(tot) ≠ DoSomethingWith(tot.wait())).

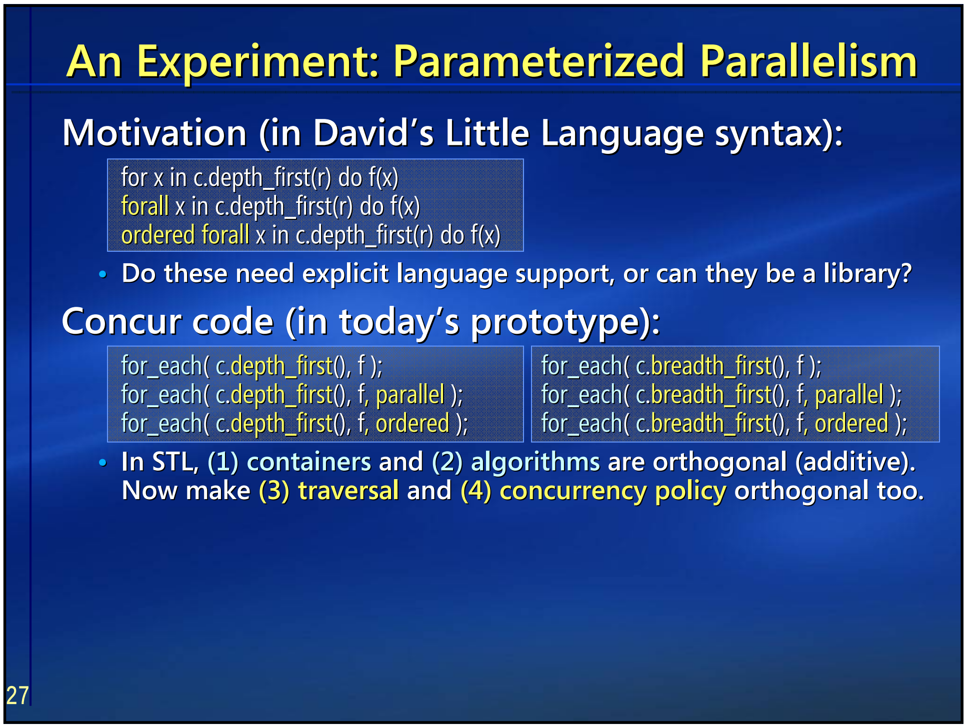


An Experiment: Parameterized Parallelism

Motivation (in David's Little Language syntax):

```
for x in c.depth_first(r) do f(x)
forall x in c.depth_first(r) do f(x)
ordered forall x in c.depth_first(r) do f(x)
```

- Do these need explicit language support, or can they be a library?



An Experiment: Parameterized Parallelism

Motivation (in David's Little Language syntax):

```
for x in c.depth_first(r) do f(x)
forall x in c.depth_first(r) do f(x)
ordered forall x in c.depth_first(r) do f(x)
```

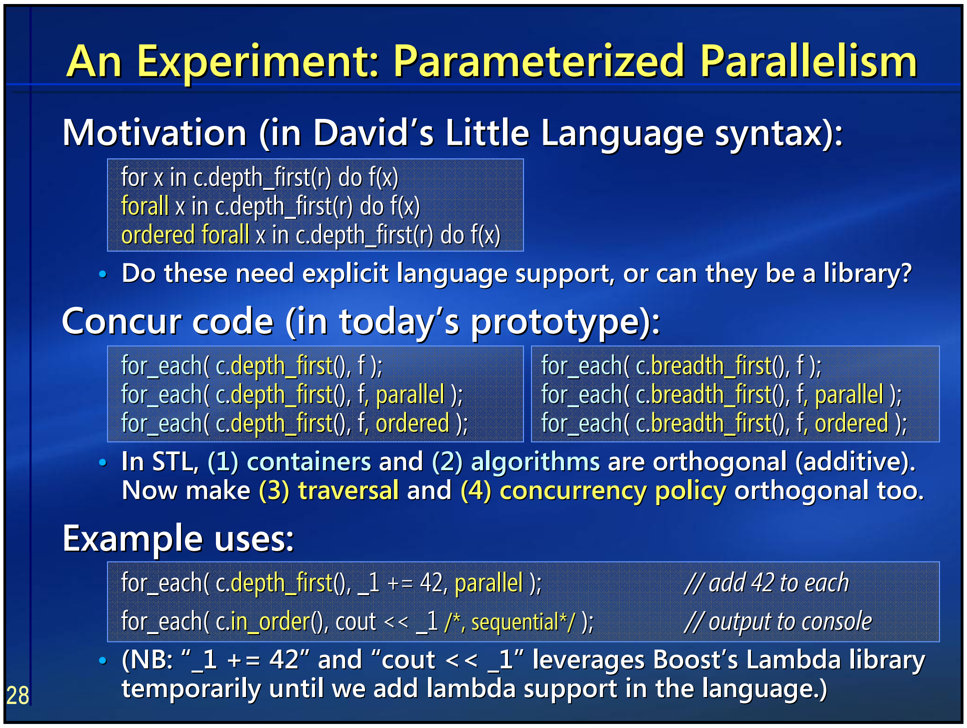
- Do these need explicit language support, or can they be a library?

Concur code (in today's prototype):

```
for_each( c.depth_first(), f );
for_each( c.depth_first(), f, parallel );
for_each( c.depth_first(), f, ordered );
```

```
for_each( c.breadth_first(), f );
for_each( c.breadth_first(), f, parallel );
for_each( c.breadth_first(), f, ordered );
```

- In STL, (1) containers and (2) algorithms are orthogonal (additive). Now make (3) traversal and (4) concurrency policy orthogonal too.



An Experiment: Parameterized Parallelism

Motivation (in David's Little Language syntax):

```
for x in c.depth_first(r) do f(x)
forall x in c.depth_first(r) do f(x)
ordered forall x in c.depth_first(r) do f(x)
```

- Do these need explicit language support, or can they be a library?

Concur code (in today's prototype):

```
for_each( c.depth_first(), f );
for_each( c.depth_first(), f, parallel );
for_each( c.depth_first(), f, ordered );
```

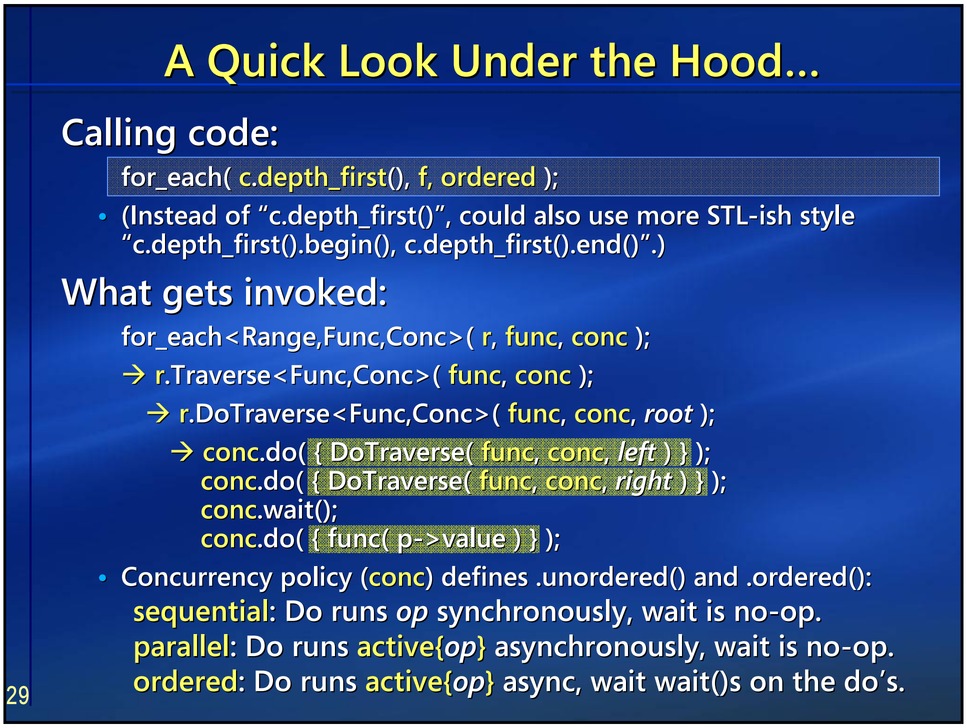
```
for_each( c.breadth_first(), f );
for_each( c.breadth_first(), f, parallel );
for_each( c.breadth_first(), f, ordered );
```

- In STL, (1) containers and (2) algorithms are orthogonal (additive). Now make (3) traversal and (4) concurrency policy orthogonal too.

Example uses:

```
for_each( c.depth_first(), _1 += 42, parallel );             // add 42 to each
for_each( c.in_order(), cout << _1 /*, sequential*/ );       // output to console
```

- (NB: "_1 += 42" and "cout << _1" leverages Boost's Lambda library temporarily until we add lambda support in the language.)



A Quick Look Under the Hood…

Calling code:

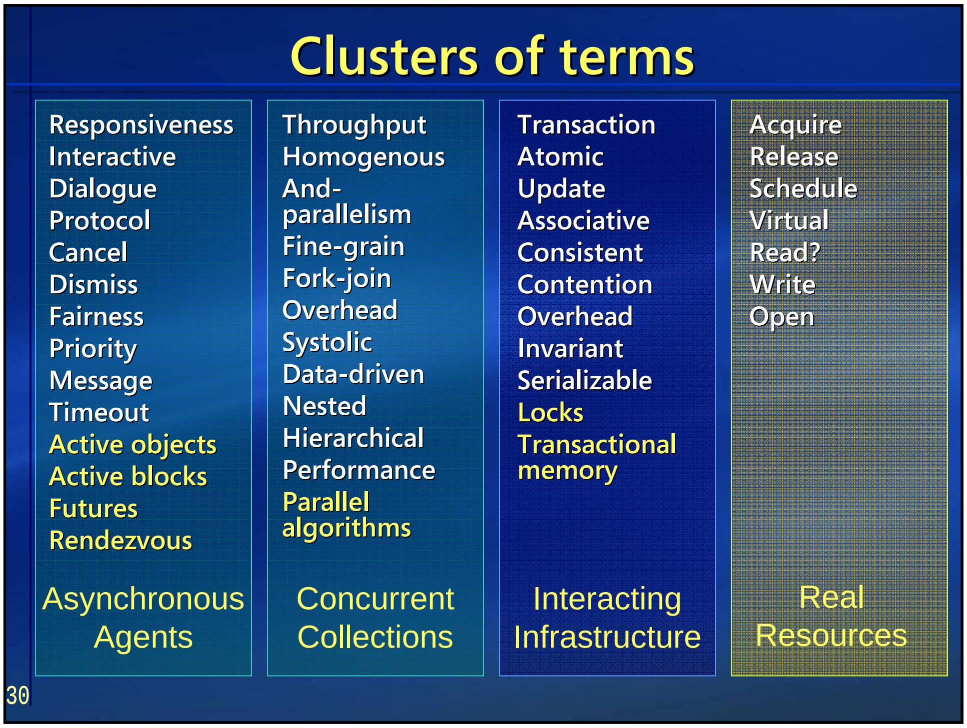
```
for_each( c.depth_first(), f, ordered );
```

- (Instead of "c.depth_first()", could also use more STL-ish style "c.depth_first().begin(), c.depth_first().end()".)

What gets invoked:

```
for_each<Range,Func,Conc>( r, func, conc );
→ r.Traverse<Func,Conc>( func, conc );
  → r.DoTraverse<Func,Conc>( func, conc, root );
    → conc.do( { DoTraverse( func, conc, left ) } );
      conc.do( { DoTraverse( func, conc, right ) } );
      conc.wait();
      conc.do( { func( p->value ) } );
```

- Concurrency policy (conc) defines .unordered() and .ordered():
  sequential: Do runs *op* synchronously, wait is no-op.
  parallel: Do runs active{*op*} asynchronously, wait is no-op.
  ordered: Do runs active{*op*} async, wait wait()s on the do's.



Clusters of terms

- Responsiveness
- Interactive
- Dialogue
- Protocol
- Cancel
- Dismiss
- Fairness
- Priority
- Message
- Timeout
- Active objects
- Active blocks
- Futures
- Rendezvous

Asynchronous Agents

- Throughput
- Homogenous
- And-parallelism
- Fine-grain
- Fork-join
- Overhead
- Systolic
- Data-driven
- Nested
- Hierarchical
- Performance
- Parallel algorithms

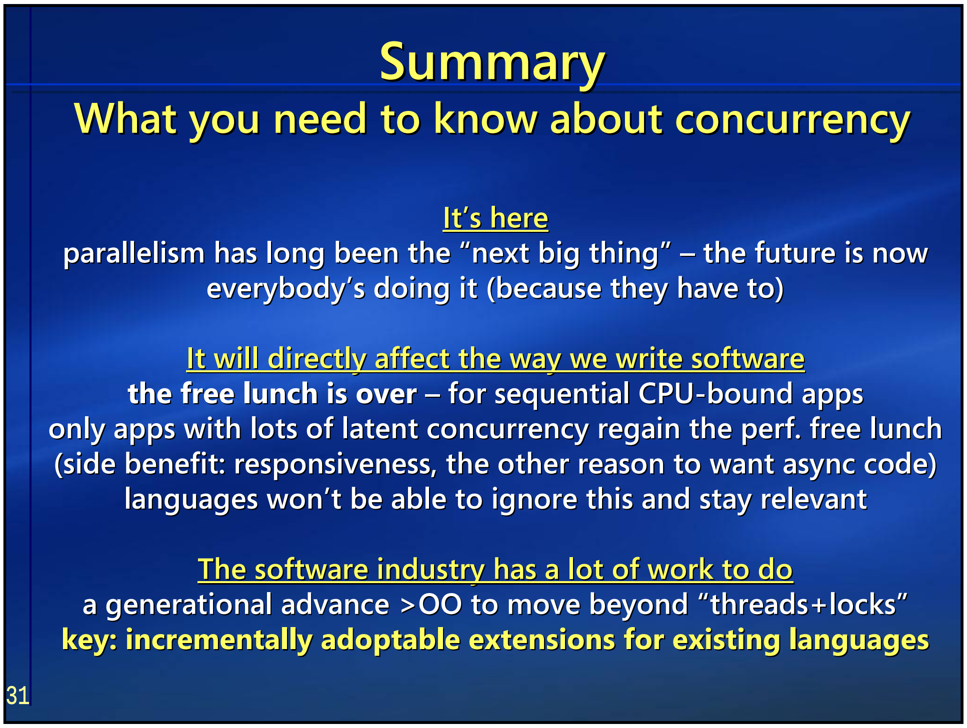
Concurrent Collections

- Transaction
- Atomic
- Update
- Associative
- Consistent
- Contention
- Overhead
- Invariant
- Serializable
- Locks
- Transactional memory

Interacting Infrastructure

- Acquire
- Release
- Schedule
- Virtual
- Read?
- Write
- Open

Real Resources



Summary

What you need to know about concurrency

It's here

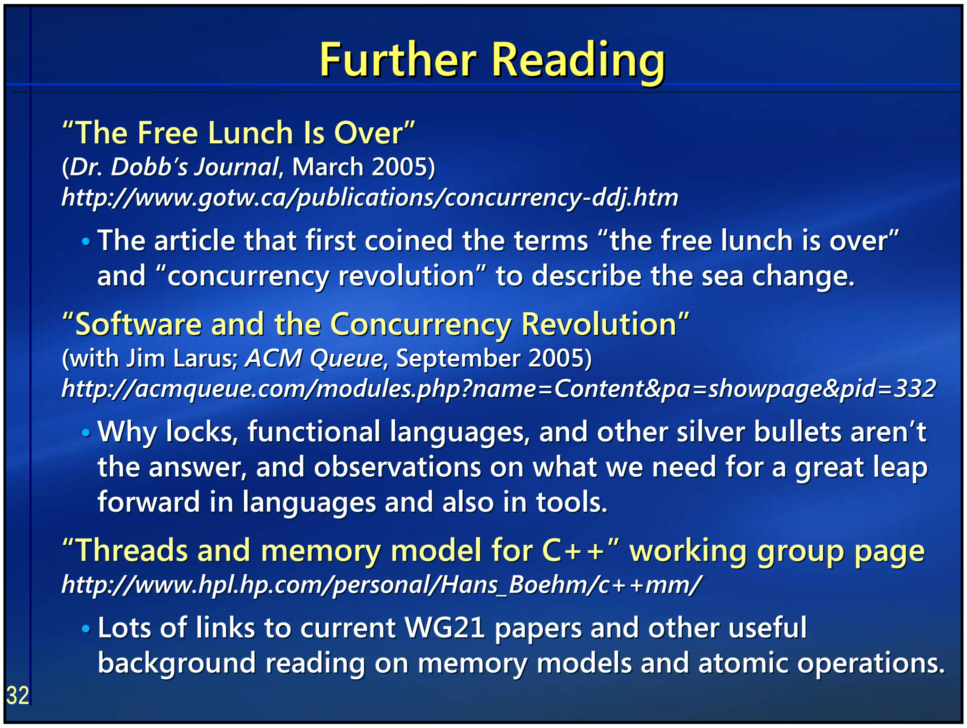
parallelism has long been the "next big thing" – the future is now
everybody's doing it (because they have to)

It will directly affect the way we write software

the free lunch is over – for sequential CPU-bound apps
only apps with lots of latent concurrency regain the perf. free lunch
(side benefit: responsiveness, the other reason to want async code)
languages won't be able to ignore this and stay relevant

The software industry has a lot of work to do

a generational advance >OO to move beyond "threads+locks"
key: incrementally adoptable extensions for existing languages



Further Reading

"The Free Lunch Is Over"
(*Dr. Dobb's Journal*, March 2005)
http://www.gotw.ca/publications/concurrency-ddj.htm

- The article that first coined the terms "the free lunch is over" and "concurrency revolution" to describe the sea change.

"Software and the Concurrency Revolution"
(with Jim Larus; *ACM Queue*, September 2005)
http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=332

- Why locks, functional languages, and other silver bullets aren't the answer, and observations on what we need for a great leap forward in languages and also in tools.

"Threads and memory model for C++" working group page
http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/

- Lots of links to current WG21 papers and other useful background reading on memory models and atomic operations.