Software and the Concurrency Revolution

Herb Sutter

Software Architect
Microsoft Developer Division

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 it 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

1

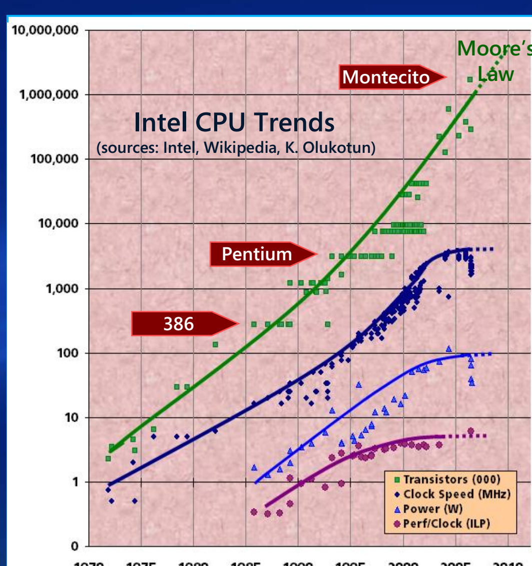
Truths

Consequences

Futures

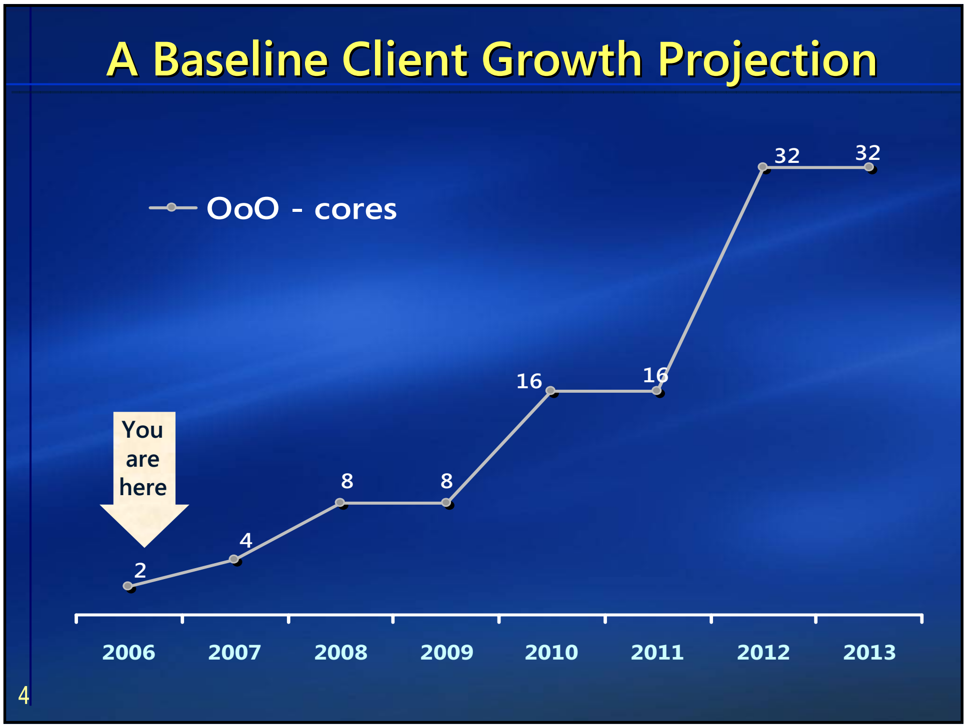
2

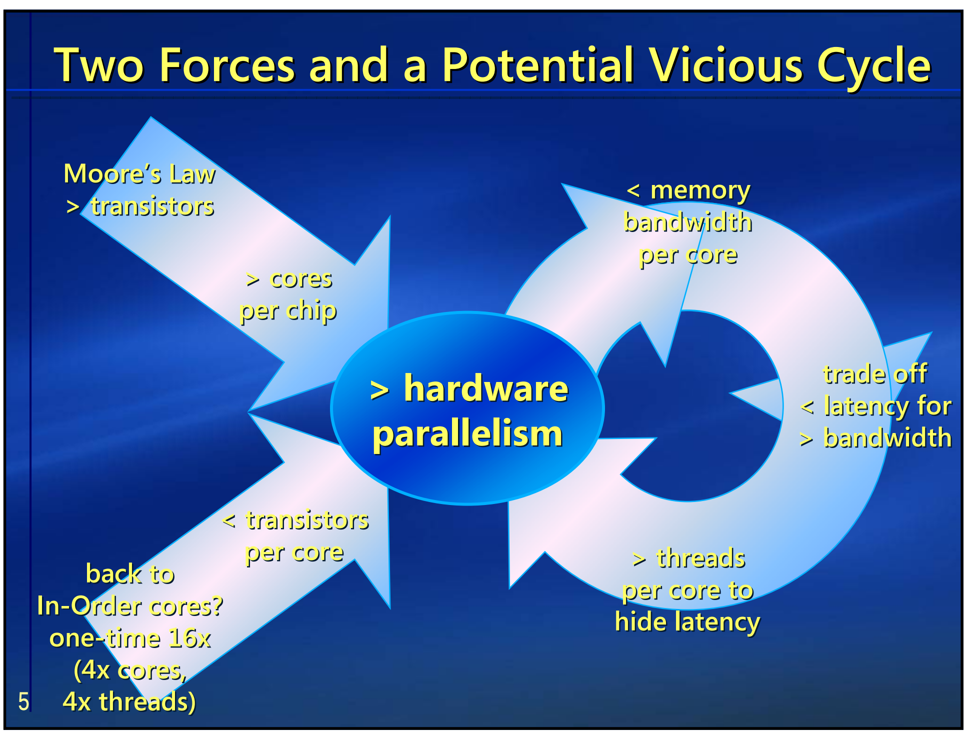
Each year we get ~~faster~~ **more** processors

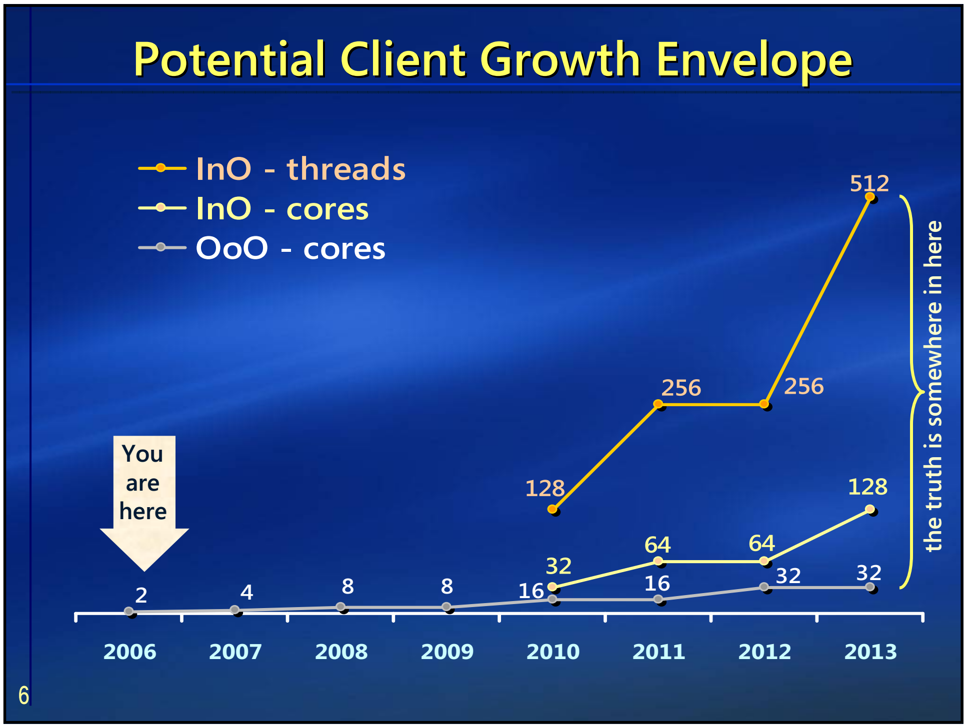


- Historically: Boost single-stream performance via more complex chips, first via one big feature, then via lots of smaller features.
- Now: Deliver more cores per chip.
- The free lunch is over for today's sequential apps and many concurrent apps (expect some regressions). We need killer apps with lots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

3


A Baseline Client Growth Projection
OoO - cores
You are here
2
4
8
8
16
16
32
32
2006
2007
2008
2009
2010
2011
2012
2013
4


Two Forces and a Potential Vicious Cycle
Moore's Law
> transistors
> cores
per chip
> hardware
parallelism
< transistors
per core
back to
In-Order cores?
one-time 16x
(4x cores,
4x threads)
< memory
bandwidth
per core
trade off
< latency for
> bandwidth
> threads
per core to
hide latency
5



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 (i.e., not Photoshop)

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)

7

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 (in some form)
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)

8

Problem 1 (of 2): **Threads**

Problem: Unstructured free threading.

- Unconstrained. Arbitrary reentrancy, blocking, affinity.

Today: Mitigate by (often) hand-coded patterns.

- Use messages (and variants, e.g., pipelines):
  Clearer and easier to reason about successfully.
- Use work queues: Manual decomposition of work + rightsized thread pool, sometimes semiautomated (e.g., BackgroundWorker).

Tomorrow:

- Enable better abstractions:
  - Active objects with implicit messages.
  - Futures.
- ("Don't roll your own vtables.")

9

Problem 2 (of 2): **Locks**

Problem: Unstructured mutable shared state.

- No composable solution for synchronizing access.

Today: Use locks. (Where are the transactions?)

“Bohr”

- Locks are best we have, but known to be inadequate:
  - Most programmers who think they know how to use locks only *think* they know how to use locks. Priesthoods abound. Even major frameworks tend to be broken.
  - Not composable.

“Quantum”

- Lock-free is sometimes applicable, but isn’t the answer:
  - Hard for geniuses to get right. A new lock-free data structure is a publishable result (often with corrections).
  - Very limited. Some basic data structures have no known lock-free implementations.
  - Helps by giving users something they don’t need to lock.

10

Problem 2 (of 2): **Locks**

Problem: Unstructured mutable shared state.

- No composable solution for synchronizing access.

Tomorrow: Greatly reduce locks. (Alas, not “eliminate.”)

1. Enable transactional programming: Transactional memory is our best hope. **Composable** `atomic` { … } blocks. Naturally enables speculative execution. (The elephant: Allowing I/O. The Achilles’ heel: Some resources are not transactable.)
2. Abstractions to reduce “shared”:
   Messages. Futures. Private data (e.g., active objects).
3. Techniques to reduce “mutable”:
   Immutable objects. Internally versioned objects.
4. Some locks will remain. Let the programmer declare:
   (1) Which shared objects are protected by which locks.
   (2) Lock hierarchies (caveat: also not composable).

11

Some Lead Bullets (useful, but mostly mined)

Automatic parallelization (e.g., compilers, ILP):

- Limited: Sequential programs tend to be… well, sequential.
- Requires accurate program analysis: Challenging for simple languages (Fortran), intractable for languages with pointers.
- Doesn't actually shield programmers from having to know about concurrency.

Functional languages:

- Contain natural parallelism… except it's too fine-grained.
- Use pure immutable data… except those in commercial use.
- Not known to be adoptable by mainstream developers.
- Borrow some key abstractions/styles from these languages (e.g., lambdas) and support them in imperative languages.

OpenMP et al.:

- "Industrial-strength duct tape," but useful where applicable.

12

A Final Word on "Truths"

Don't underestimate the programming problem.

The hardware community is building parallel hardware,
but do you recognize how hard it is to program?

Don't assume the guy upstream
can and will solve the hard problems.

This talk has mentioned ideas on future software directions,
but these aren't (yet) proven solutions or shipping products.

Hardware semantics and operations should
focus on programmability first, speed second.

In particular, non-sequentially consistent memory models
are an enormous source of difficulty for programmers.

See for example "Multiprocessors Should Support Simple Memory Consistency Models," Mark D. Hill, IEEE Computer, August 1998. Affirmed at Dagstuhl 2003.

Software can help mitigate: Try to keep both SC and performance
by reducing/eliminating mutable shared state. (Easy to say…)

13

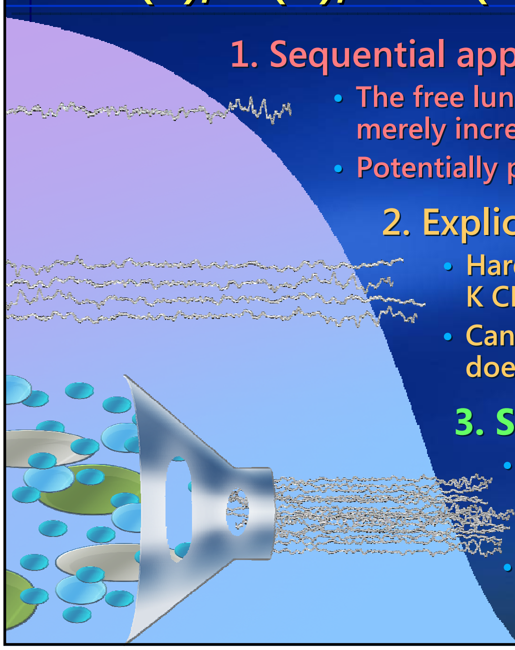
Truths

Consequences

Futures

14

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)

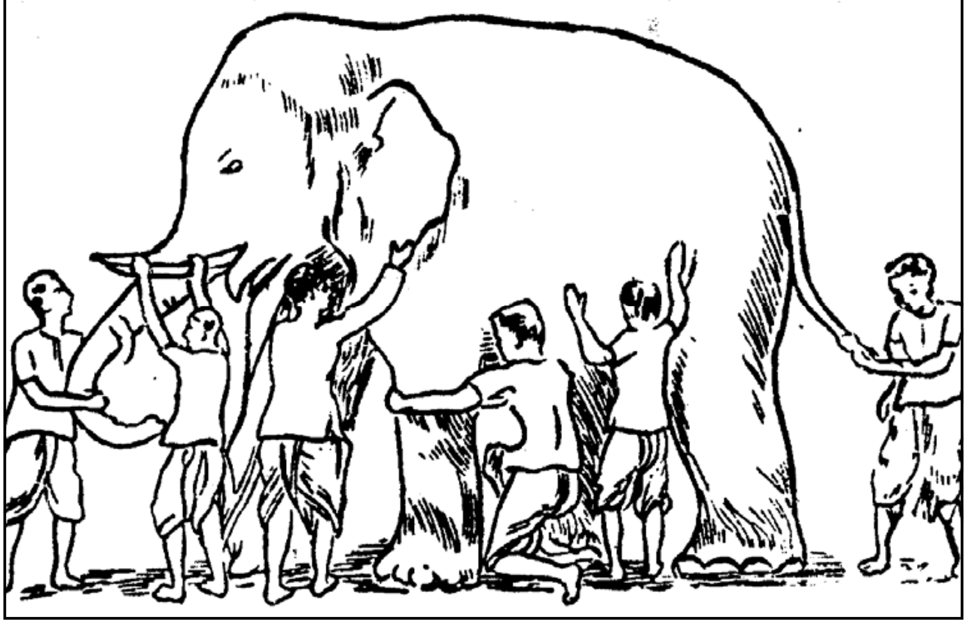
3. Scalable concurrent apps.
   - Workload decomposed into a "sea" of heterogeneous work items (with ordering edges).
   - Lots of latent concurrency we can map down to N cores.

An OO for Concurrency

OO	
Fortran, C, …	threads+locks
asm	semaphores

17

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

19

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	Locks	
	Performance		

20

Toward an "OO for Concurrency"

Lots of work across the stack, from App to HW

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, …)

21

Truths

Consequences

Futures

22

Concurrency Tools in 2006 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:

- ISO C++: 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, …).

23

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.**

24

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, ...

25

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.

26

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()
```

27

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())).

28

Using Futures and Active Lambdas

Active blocks (lambdas) for queueing up work items:

```
x = active { foo(10) };      // call foo asynchronously
y = active { a->b( c ) };    // evaluate asynchronously
p = active { new T };        // allocate and construct asynchronously
... more code, runs concurrently with all three active lambdas ...
return x.wait() * y.wait() * p.wait()->bar();
```

Idioms:

- "Active" to call a sync function async, or get outside locks:

```
active { plainObj.Foo(42) }        // type is future<ReturnType>
```

- "Wait" to call an async function synchronously:

```
   activeObj.Bar(3.14).wait();     // type is ReturnType
or wait( activeObj.Bar(3.14) );
```

- "Active...wait" to get outside locks and leave caller interruptible:

```
active { SomeLongOperation() }.wait();
```

- "Active" to do something later when a future is ready:

```
active { int i = f.wait(); DoSomethingWith( i );  /*...*/  }
```

29

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?

30

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 );
```

31

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.

32

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
```

33

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	Locks (declarative support for)	
Active objects	Performance	Transactional memory	
Active blocks	Parallel algorithms		
Futures			
Rendezvous			

34

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 it 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

35

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 used 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.

36

Questions?