The Concur Project

Some Experimental Abstractions for Imperative Languages

Herb Sutter
Software Architect
Microsoft Developer Division

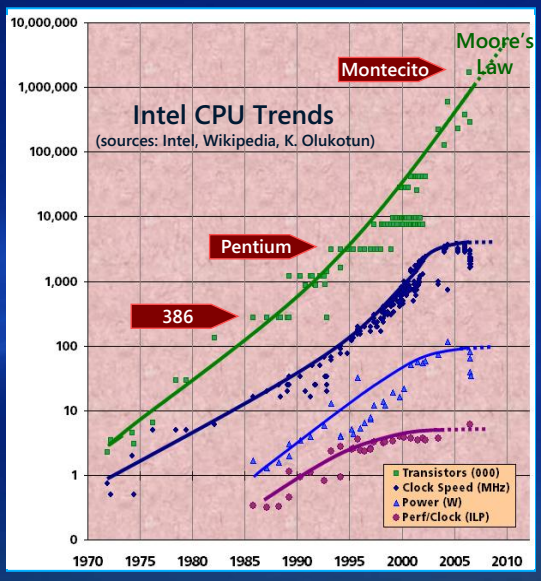
Truths

Consequences

Futures

1

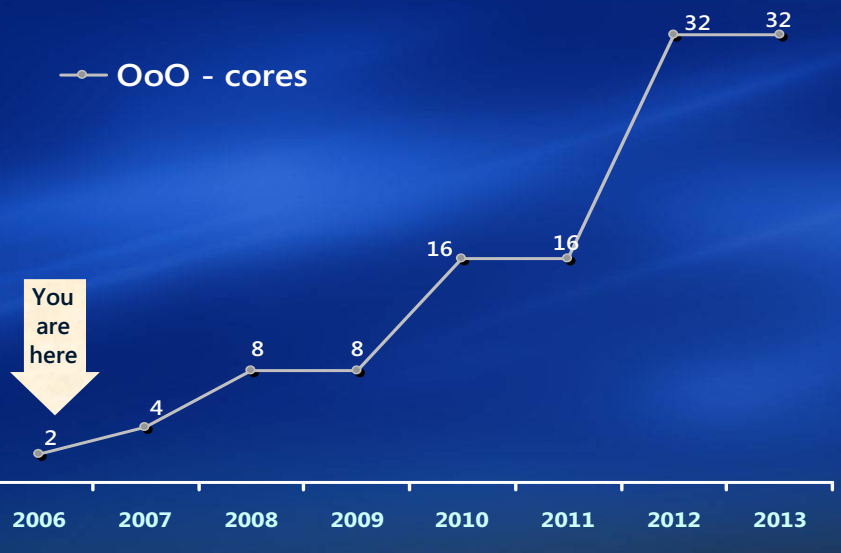
Each year we get ~~faster~~ more processors



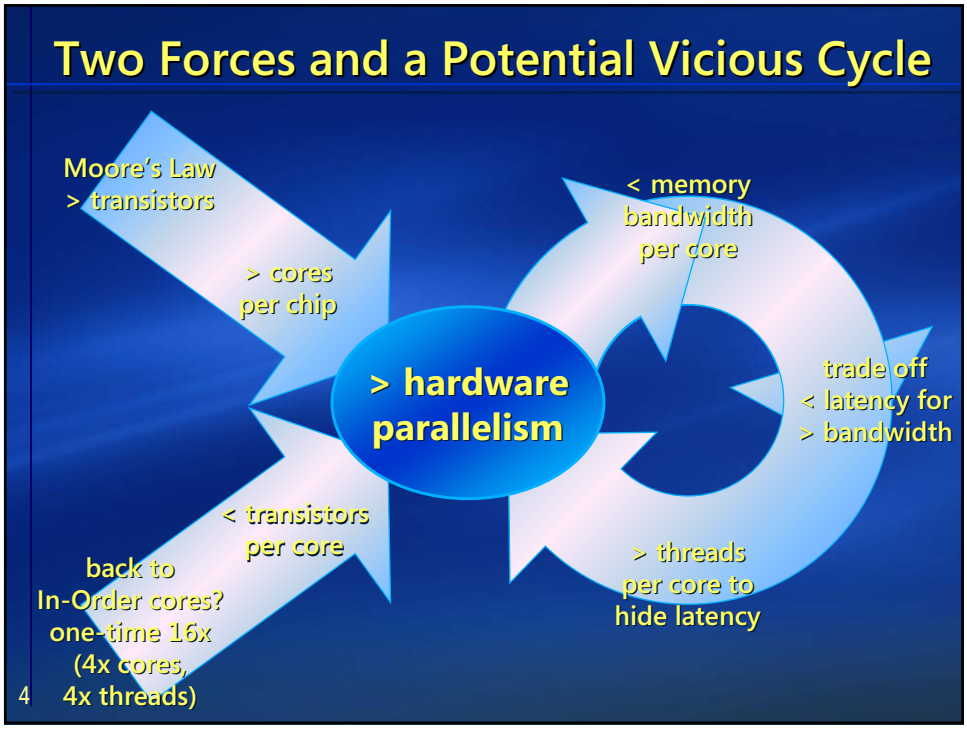
- Historically: Boost single-stream performance via more complex chips, first via one big feature, then via lots of smaller features.
- Now: Deliver more cores per chip.
- The free lunch is over for today's sequential apps and many concurrent apps (expect some regressions). We need killer apps with lots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

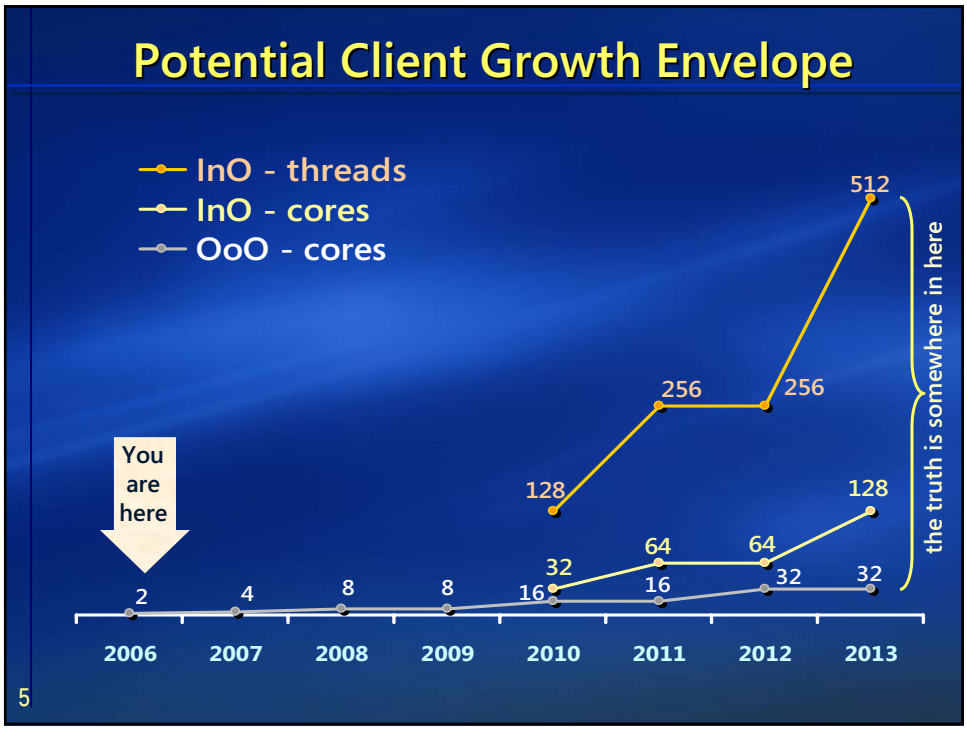
2

A Baseline Client Growth Projection



3


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


Potential Client Growth Envelope
InO - threads
InO - cores
OoO - cores
You
are
here
2
4
8
8
16
32
64
128
256
256
16
32
64
128
512
2006
2007
2008
2009
2010
2011
2012
2013
the truth is somewhere in here
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)

6

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.")

7

Problem 2 (of 2): Locks

Problem: Unstructured mutable shared state.

- No composable solution for synchronizing access.

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

- 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.
- 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.

"Bohr"

"Quantum"

8

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).

9

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.

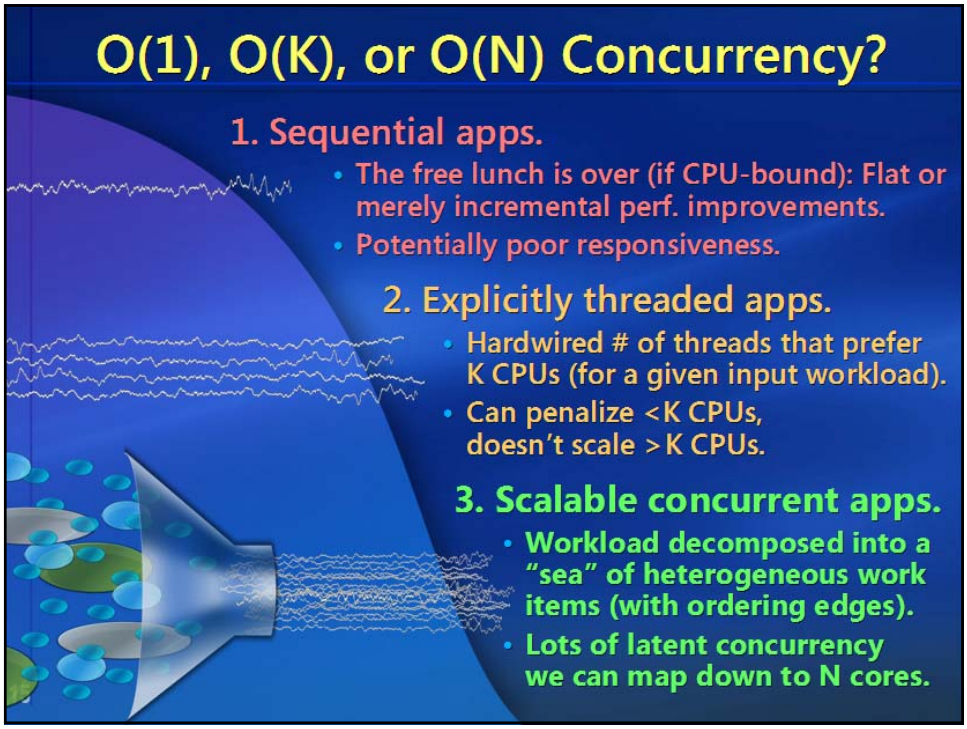
10

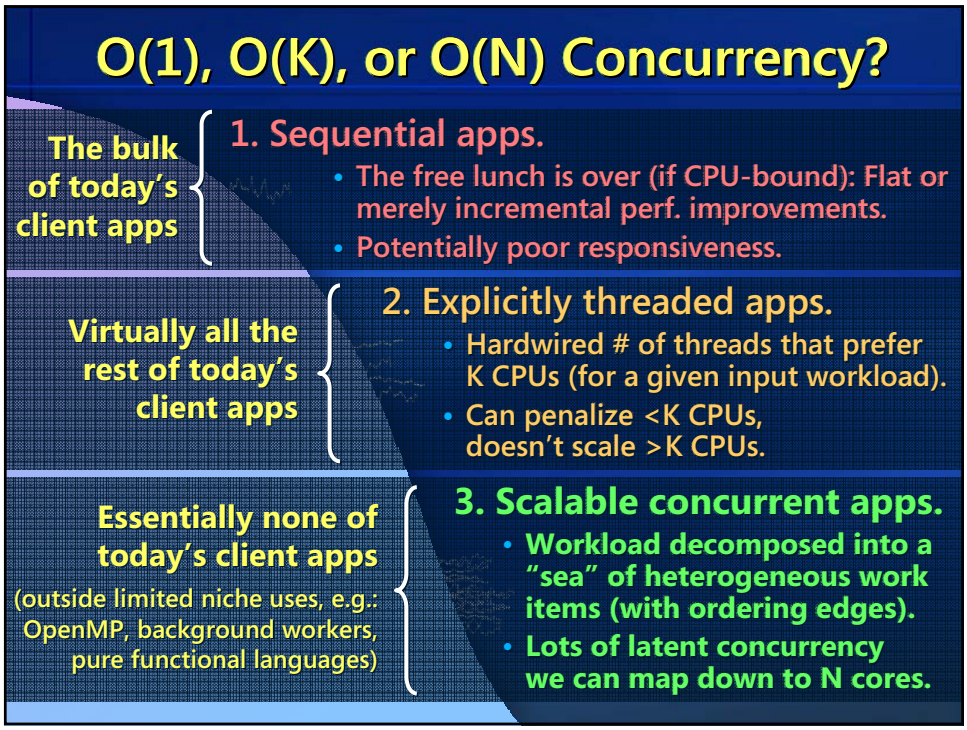
Truths

Consequences

Futures

11


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.

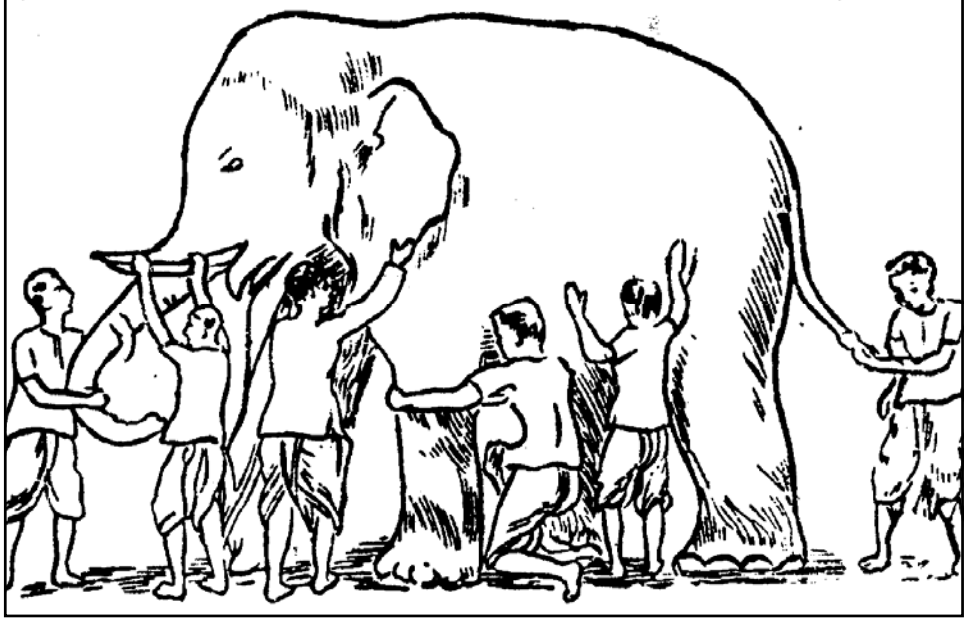

O(1), O(K), or O(N) Concurrency?
The bulk of today's client apps
1. Sequential apps.
• The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
• Potentially poor responsiveness.
Virtually all the rest of today's client apps
2. Explicitly threaded apps.
• Hardwired # of threads 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

14

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		

16

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		

17

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, …)

18

Truths

Consequences

Futures

19

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, …).

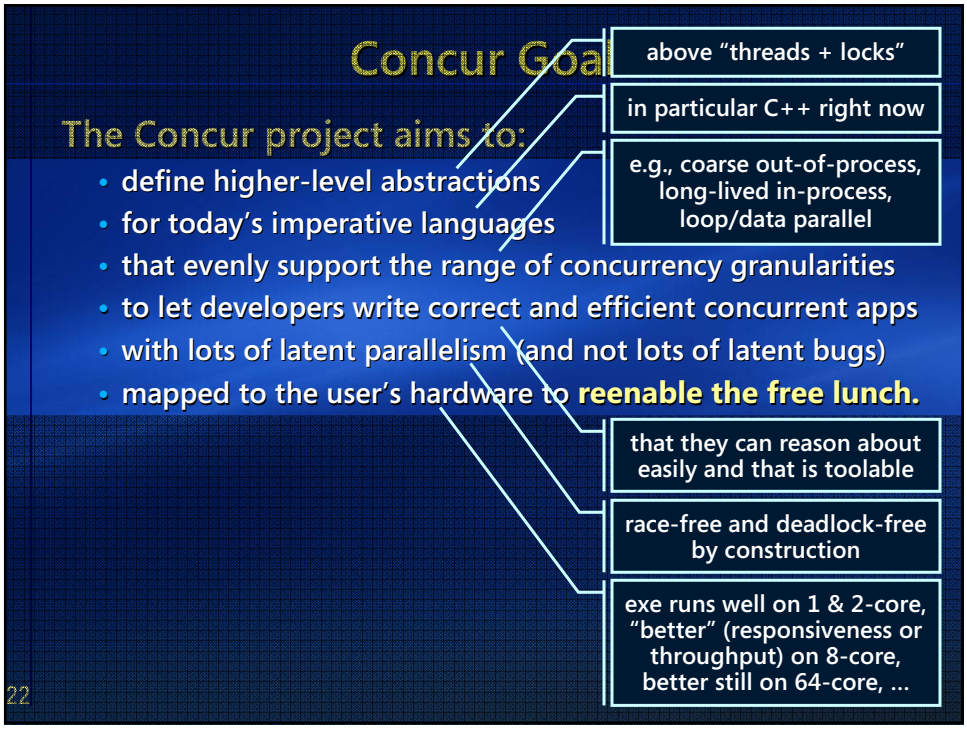
20

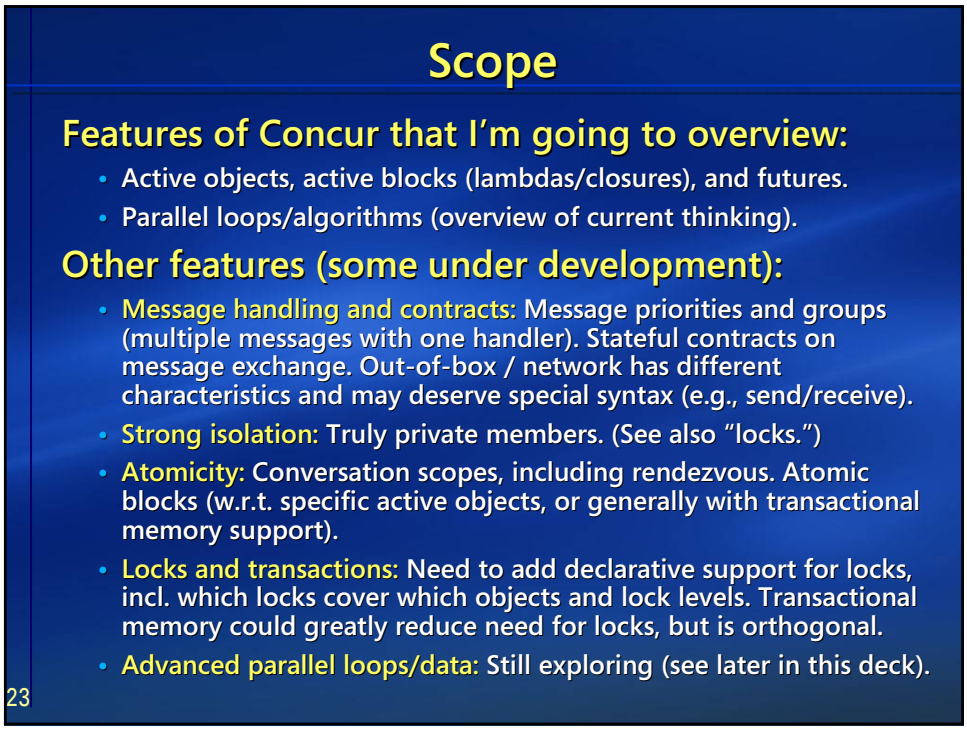
Concur Goals

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.**

21


Concur Goal
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, ...
22


Scope
Features of Concur that I'm going to overview:
• Active objects, active blocks (lambdas/closures), and futures.
• Parallel loops/algorithms (overview of current thinking).
Other features (some under development):
• Message handling and contracts: Message priorities and groups (multiple messages with one handler). Stateful contracts on message exchange. Out-of-box / network has different characteristics and may deserve special syntax (e.g., send/receive).
• Strong isolation: Truly private members. (See also "locks.")
• Atomicity: Conversation scopes, including rendezvous. Atomic blocks (w.r.t. specific active objects, or generally with transactional memory support).
• Locks and transactions: Need to add declarative support for locks, incl. which locks cover which objects and lock levels. Transactional memory could greatly reduce need for locks, but is orthogonal.
• Advanced parallel loops/data: Still exploring (see later in this deck).
23

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.

24

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()
```

25

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())).

26

Design Alternative: Implicit Futures

Mort (in particular) might prefer implicit futures:

- A common case is for futures that: a) are local, and b) don't need to have future<T> methods called (e.g., Cancel).

```
double tot = calc.TotalOrders();  // call is nonblocking
... potentially lots of work ...  // parallel work
DoSomethingWith( tot );           // implicitly wait to accept
```

Issues:

- Can silently lose concurrency (e.g., early "WriteLine(tot);"), but Mort is still getting extra concurrency for free.
- Implicit block point for emitting exceptions.
- Passing futures onward to other code (e.g., calling DoSomethingWith(tot) that is overloaded on double and future<double>... no disambiguation?).
- Requires some type system games ("exactly what type is it?").

27

Comparison: FX Async Pattern

Design Guidelines base async pattern:

- Nearly identical to Outlook's native version.
- Split single calls into BeginXxx/EndXxx pairs with intermediate structures and explicit callback registrations.
- Sync API:

```
public RetType MethodName(
  Parameters params,
  OutParameters outParams );
```

- Async API:

```
IAsyncResult BeginMethodName(// caller explicitly calls begin/end
  Parameters params,
  AsyncCallback callback,          // caller must supply callback
  Object state );                  // caller usu. must supply cookie
RetType EndMethodName(             // caller explicitly calls begin/end
  IAsyncResult asyncResult,
  OutParameters outParams );
```

28

Comparison: FX Async Pattern (2)

Issues:

- Intrusive in API: Burying the async work partly inside the APIs changes/bloats the APIs and can duplicate code.
- Complex for callers: Instead of calling one function, the user must call two functions, manage intermediate state, and write an additional function of his own for the callback.
- Hand-coded pattern, no language support/checking: The pattern may vary; consistency depends on manual discipline. This approach has more potential points of failure where the programmer can go wrong.
  - DG example: *"Do ensure that the EndMethodName method is always called even if the operation is known to have already completed. This allows exceptions to be received and any resources used in the async operation to be freed."*

29

Example: Multiple Concurrent Requests

Problem:

- Process calls Work1, Work2, and Work3, all concurrently.
- Process computes the result of Work1 plus the result of either Work2 or Work3, whichever is available first (Work2 and Work3 are redundant requests, e.g., different servers, different algos).
- Process takes a timeout, and if the result cannot be computed before the timeout expires, Process returns -1.
- Process should not busywait.

```
// Pseudocode
int Process( Service1 s1, Service2 s2, Service3 s3, int ms ) {
  int result1 = s1.Work1( 42 );        // should run asynchronously
  int result2 = s2.Work2( "xyzzy" ); // should run asynchronously
  int result3 = s3.Work3( "xyzzy" ); // should run asynchronously
  if( Work1 and either Work2 or Work3 complete within "ms" ms ) {
    return result1 + whichever of result2 or result3 is available;
  } else {
    return -1;
  }
}
```

30

Example: Multiple Concurrent Requests

```
// Concur (alternative 2)

int Process( active Service1 s1, active Service2 s2, active Service3 s3, int ms ) {
  future<int> result1 = s1.Work1( 42 ),
              result2 = s2.Work2( "xyzzy" ),
              result3 = s3.Work3( "xyzzy" );
  future_group g = result1 && (result2 || result3);
  ( g || timeout(ms) ).wait();
  return g.IsComplete() ? result1.wait() + (result2 || result3).wait() : -1;
}
```

31

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 );   /*...*/   }
```

32

Destructor/Dispose Semantics

Calling an active object's destructor blocks the caller:

- Till object's execution ends. Only case of blocking w/o "wait."
- Join points are typically at block exits. Supports ganging.

```
{
  active C c;
  ...
} // waits for c to complete
```

- Can directly express task hierarchies (tasks that own other tasks) as composed objects by associating member lifetimes.

```
active class C {
  active M m;                    // embedded member
  ...
};
// later
{
  active C c;
  ...
} // waits for c & c.m to complete
```

33

Example: Producer-Consumer

```
active class Consumer {
  public: void Process( Message msg ) { ... }
};
active Consumer consumer;                    // runs asynchronously
```

```
active class Producer {
  public: void Produce() {
    for( int i = 0; ++i < 100; ) {
      Message msg = ... ;
      consumer.Process( msg );               // non-blocking call
    }
  }
};
```

```
int main() {
  active Producer p;                         // runs asynchronously
  p.Produce();                               // non-blocking call
} // wait for p to end
// wait for consumer to end
```

- Impact vs. nonconcurrent: Add active.

34

OpenMP and Beyond

OpenMP offers many benefits:

- Portable, scalable, flexible, standardized, and performance-oriented interface for parallelizing code.
- Hides many details: Thread team created at app startup, per-thread data allocated when #pragma entered, and work divided into coherent chunks.

But it also has many limitations:

- C, not C++ (and not C#, not Python, not ...).
- Outside the language, bolted onto code via #pragmas.
- Best suited to simple loops over arrays (doesn't work well with STL or BCL collections), and for parallel code sections.
- Doesn't take advantage of C++'s (and other langs') superior abstractions for type safety or generic programming.

35

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?

36

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 );
```

37

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.

38

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
```

39

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.

40

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			

41

Concur Summary

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.

42

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.

43


Questions?