

actors

**a unifying design pattern
for scalable concurrency**

www.iolanguage.com
steve@dekorte.com

talk overview

what is an actor?

concurrency trends

problems and solutions

the big picture

what is an actor?

an informal definition

an object with an asynchronous message queue and an execution context for processing that queue

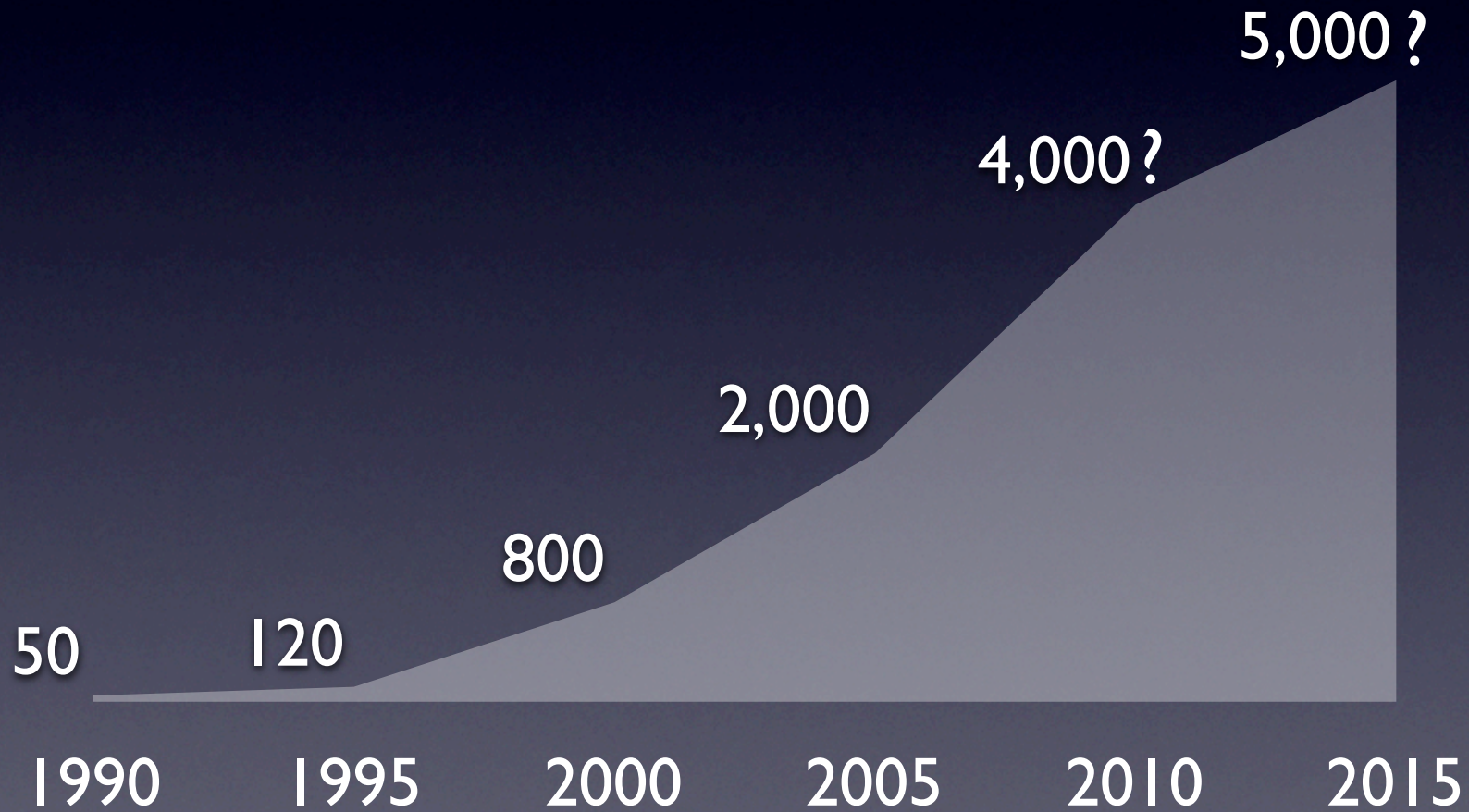
encapsulates state, instructions and execution

(a CSP is a process level actor)

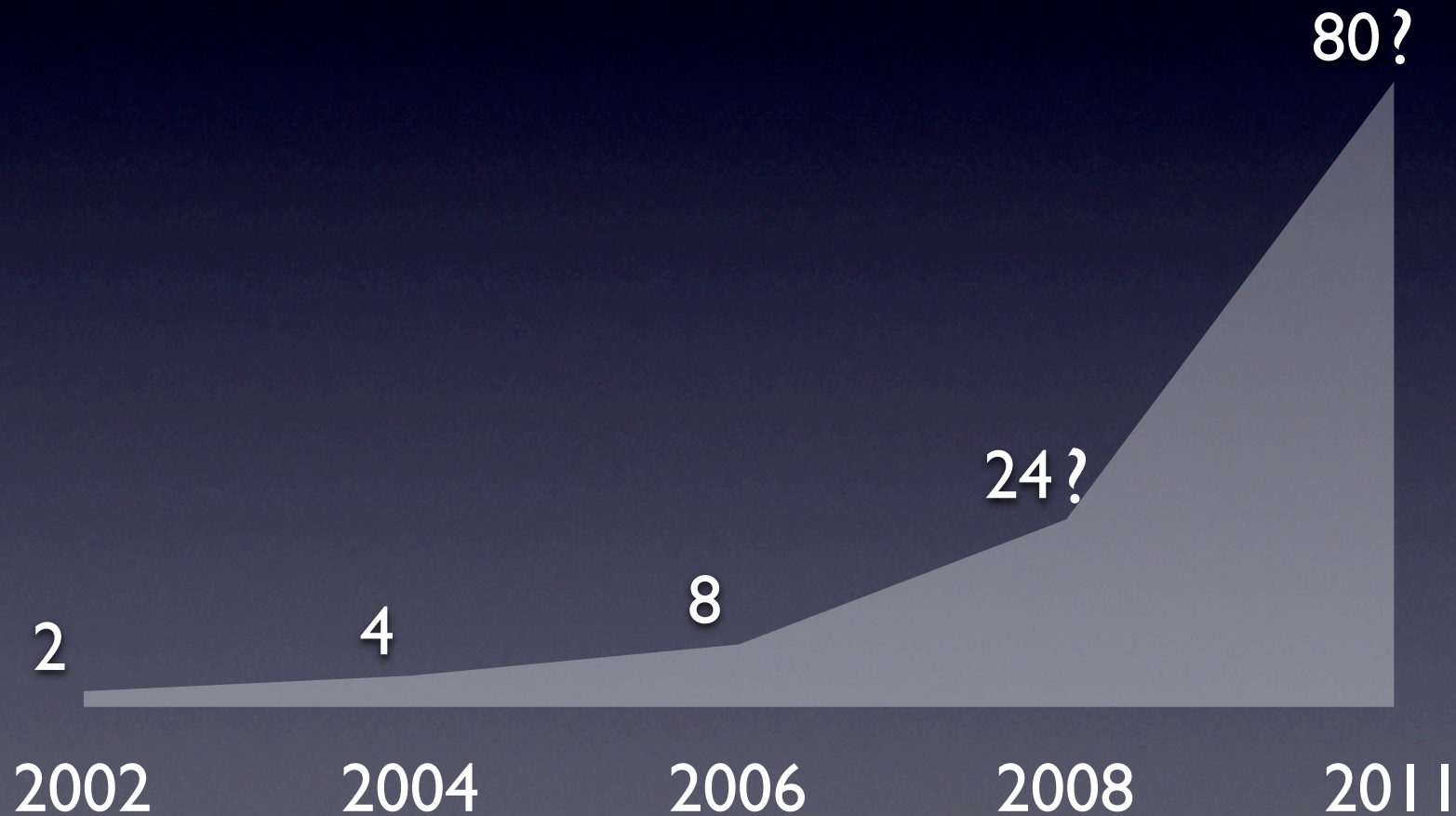
concurrency trends

a quick look

clock speed leveling off



cores per machine increasing exponentially



clusters

massive scaling

	typical scale
memcached	10^2
gfs	10^4
mmog	10^3
p2p	10^5
@home	10^6

trends are
cores and clusters

ideal concurrency model
will naturally scale across both

traditional concurrency model

preemptive threads with
shared memory and
coordination via locks

problem

nondeterminism

For concurrent programming to become mainstream, we must discard threads as a programming model.

Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

- Ed Lee, The Problem with Threads
Berkeley CS Tech Report

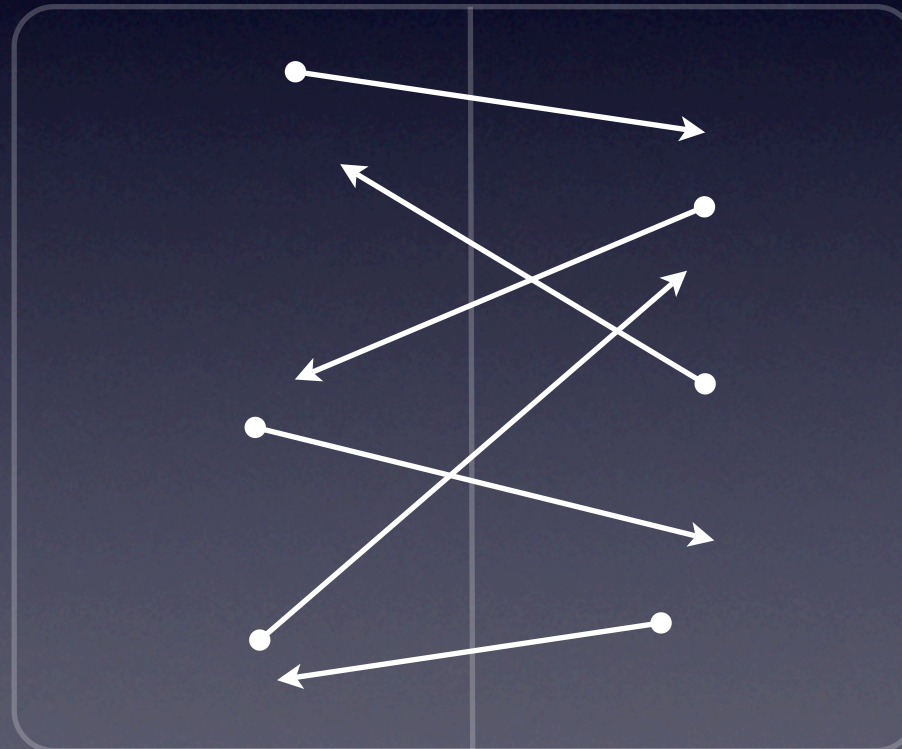
traditional concurrency model

threads can directly change one another's state

“spaghetti concurrency”

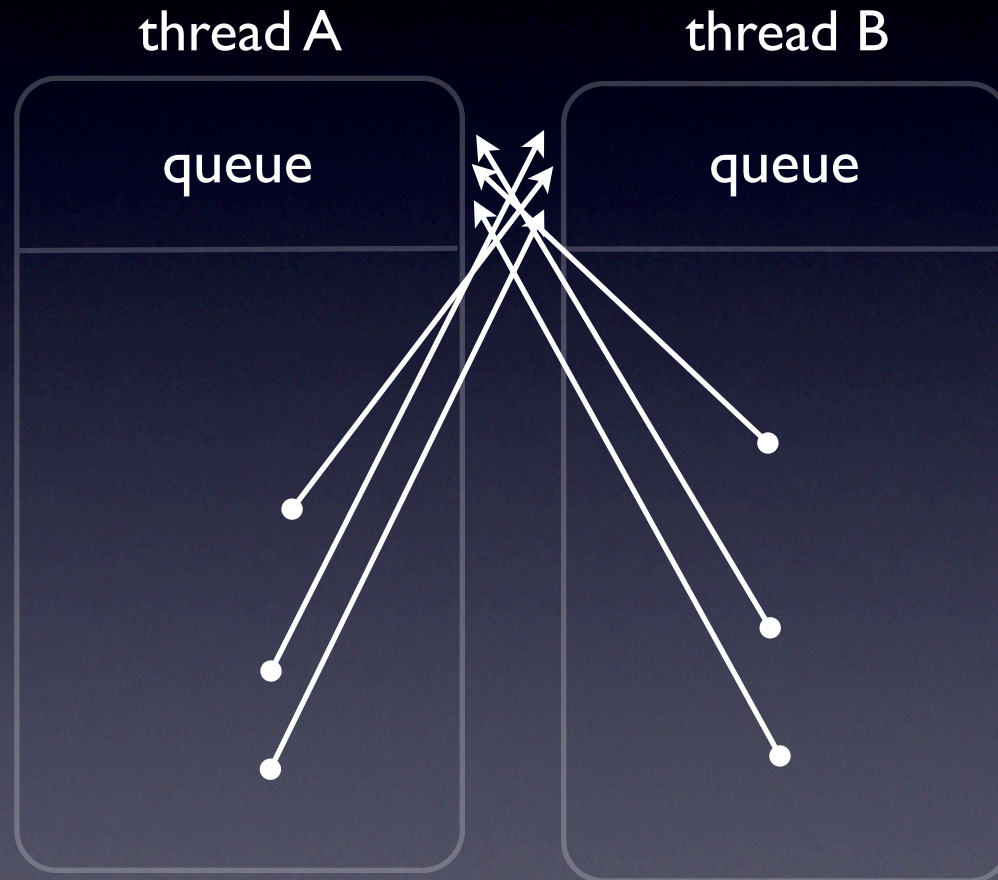
thread A

thread B



actor/csp model

only a thread can directly change it's own state



same model works across machines

a natural extension

actors are the object
paradigm extended to execution

objects encapsulate state and instructions

actors encapsulate state, instructions
and execution

what this looks like in lo

**any object becomes an actor when
sent an async message**

```
account deposit(10.00) // sync message
```

```
account @deposit(10.00) // future message
```

```
account @@deposit(10.00) // async message
```

lo is a hybrid actor language

problem

**asynchronous
programming**

a future

an object returned from an async message which becomes the result when it is available. If it is accessed before the result is ready, it blocks the calling execution context.

futures

sync programming with async messages by
decoupling messages from return values

allows lazy synchronization and
automatic deadlock detection

another look

```
account deposit(10.00) // sync message
```

```
account @deposit(10.00) // future message
```

```
account @@deposit(10.00) // async message
```

another example

```
// waits for result
```

```
data := url fetch
```

```
// returns a future immediately
```

```
data := url @fetch
```

```
data setFutureDelegate(self)
```

the undiscovered country

lots of interesting concurrent coordinating patterns can be composed from futures or other forms of async return messages

```
c := urls cursor  
c setConcurrency(500)  
c setDelegate(self)  
c foreach(fetch)
```

problem

distributed programming

transparent distributed objects

unified local and remote messaging

```
peers append(DO at(ip, port, key))
```

```
...
```

```
c := peers cursor
```

```
c setConcurrency(50)
```

```
c setDelegate(self)
```

```
c foreach(search(query))
```

eliminates protocol hassles but
doesn't mean we can ignore the network, etc

problem

**the high concurrency
memory bottleneck**

execution contexts

memory usage and maximum concurrency

	bytes	max per GB
process/os thread	1,000,000s	100s*
stackfull coro	10,000s	10,000s
stackless coro or continuation	100s	1,000,000s**

*webkit etc use os threads..

**but thread related state may exceed stack size

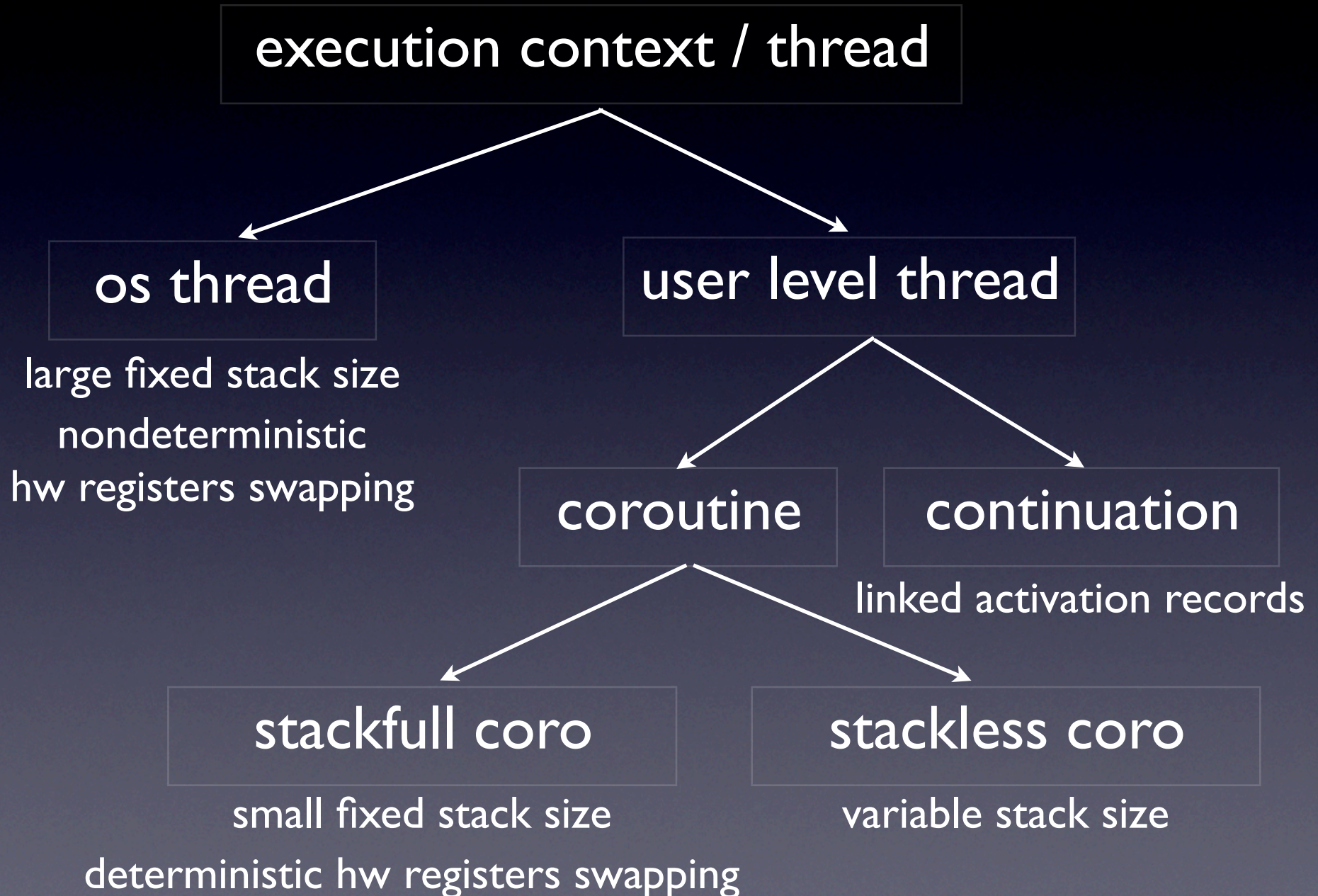
synonyms

user level thread
lightweight thread
microthread
green thread
coroutine
coro
fiber

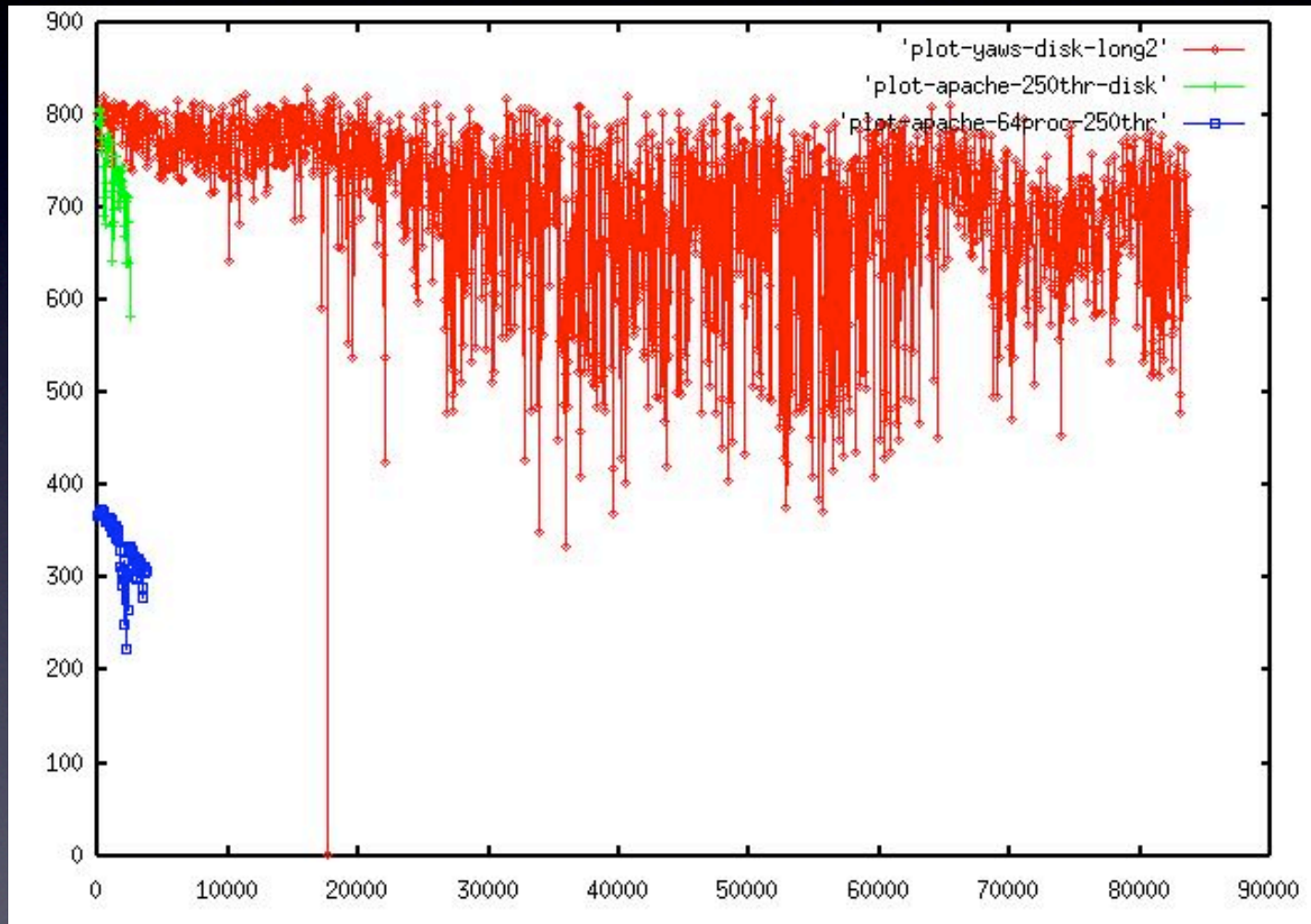
synonyms

os thread
kernel thread
native thread

taxonomy



what this means in practice



connections

user level threads aren't
preemptive, so what about
blocking ops?

avoid them by using
async sockets and
async file i/o

async issues

ease of use?

pause the user level thread while waiting on the async i/o request

cpu bound blocking?

use explicit `yield()` where needed

conclusion

use user level threads for scaling
concurrency on a given core and
one os threads or processes per
core for scaling across cores

the big picture

“powers of 10”

each level follows the actor pattern of encapsulating state, instructions and execution and communicating via async queued messaging

actor
user thread

actor

actor

actor

actor

actor

actor

csp

actor

actor

os process

actor

actor

actor

actor

actor

actor

actor

actor

csp

csp

csp

csp

csp

csp

csp

csp

machine

csp

csp

csp

csp

csp

csp

csp

csp

machine

machine

machine

machine

machine

machine

machine

machine

cluster

machine

machine

machine

machine

machine

machine

machine

machine

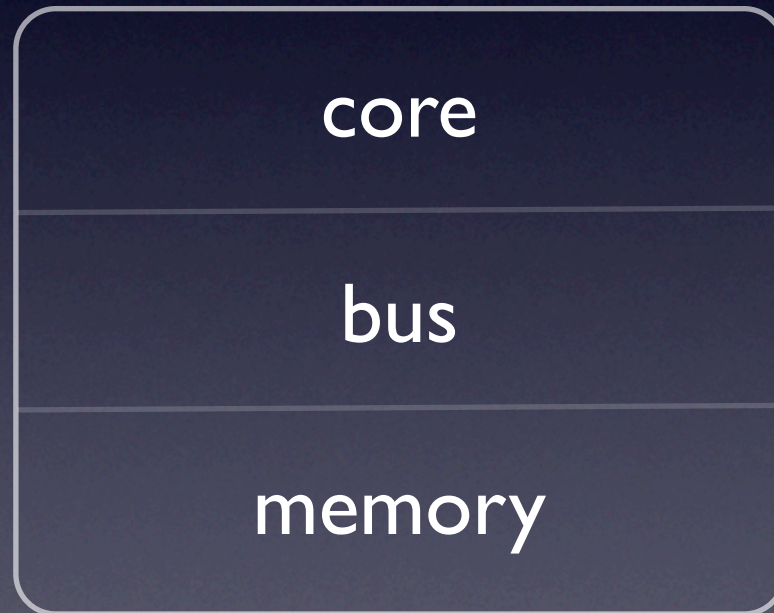
some fun speculation

what about cores?

a prediction based on this pattern

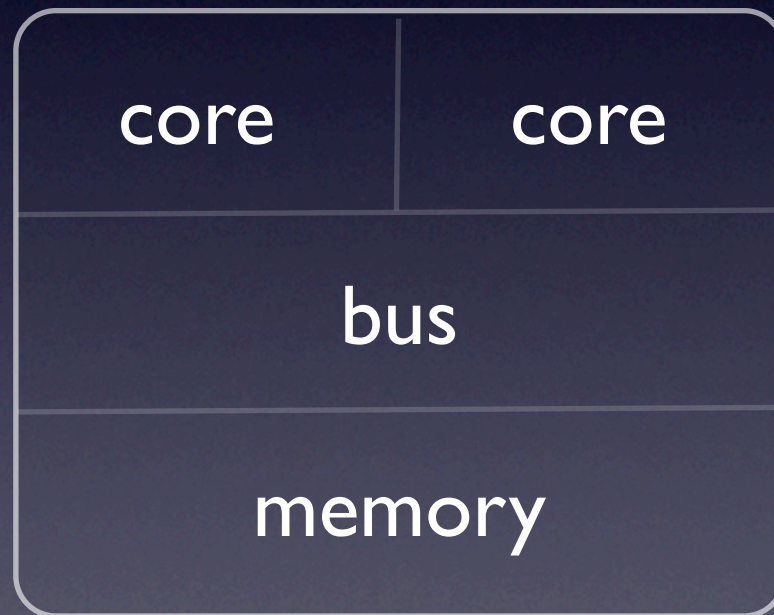
traditional SISD architecture

works, but clock speed growth is slowing
and silicon is cheap



current MISD architecture

bus bottleneck - memory performance per core drops as core count increases

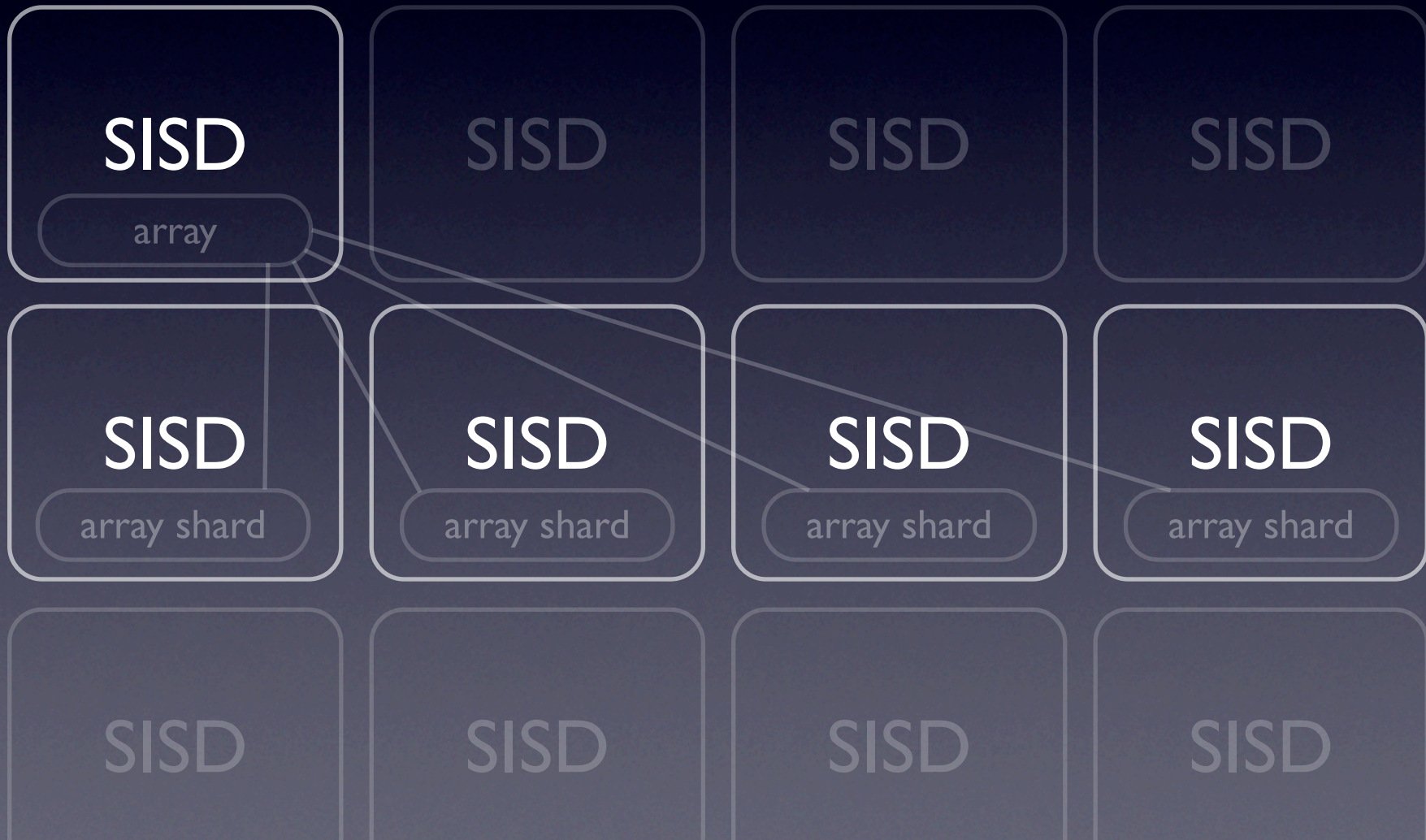


future MIMD architecture?

the actor pattern on the hardware level
a connection machine on a chip



automatic MIMD distribution



this talk, in a nutshell

solution	problem	solution
concurrency	nondeterminism	actors/csp
actors/csp	distributed and async programming	transparent distributed objects and futures
high concurrency	memory bottleneck	user level threads
many cores	bus bottleneck	MIMD

www.iolanguage.com

steve@dekorte.com