Our Concurrent Past; Our Distributed Future

Joe Duffy
Co-founder/CEO Pulumi

The world is concurrent

File and network I/O.

Servers with many users.

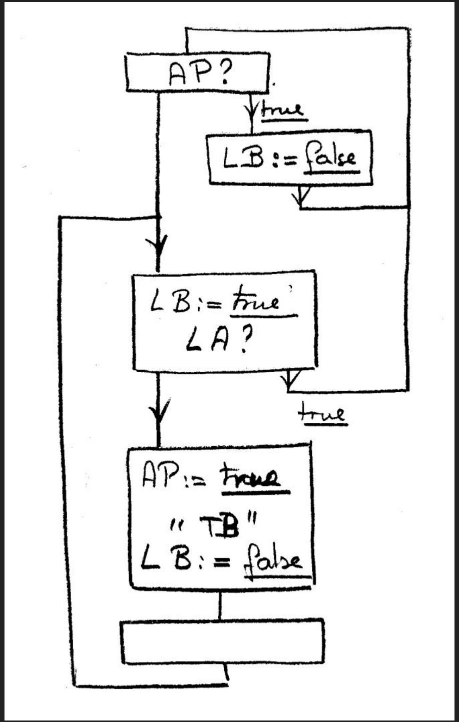
Multicore.

Mobile devices.

GPUs.

Clouds of machines.

All of which communicate with one another.



Distributed is concurrent

Concurrency: two or more things happening at once.

Parallelism is a form of concurrency: multiple workers execute concurrently, with the goal of getting a “single” job done faster.

So too is distributed: workers simply run further apart from one another:

Different processes. Different machines.

Different racks, data centers. Different planets.

Different distances implies different assumptions/capabilities.

Long lost brothers

Concurrency and distributed programming share deep roots in early CS.

Many of the early pioneers of our industry -- Knuth, Lampson, Lamport, Hoare -- cut their teeth on distributed programming.

Concurrency and parallelism broke into the mainstream first.

Now, thanks to mobile + cloud, distributed will overtake them.

By revisiting those early roots, and reflecting on concurrency, we can learn a lot about what the future of distributed programming holds for us.

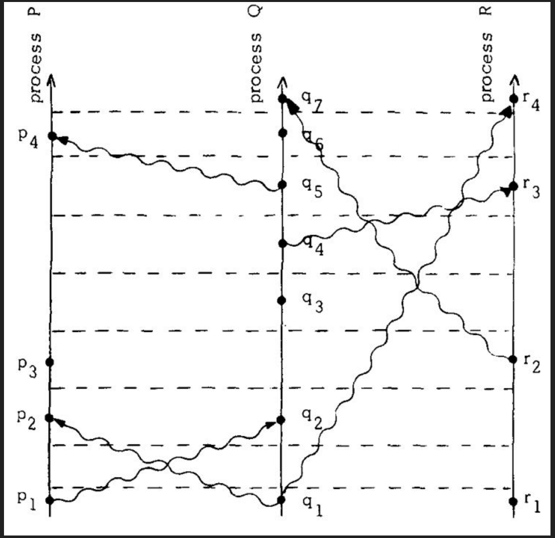
How we got here

First there was “distributed”

Early OS work modeled concurrency as distributed agents.

- Dijkstra semaphores (1962).
- Hoare CSPs (1977).
- Lamport time clocks (1978).
- Butler Lampson on system design (1983).

Even I/O devices were thought to be distributed; asynchronous with respect to the program, and they did not share an address space.



Modern concurrency roots

Many abstractions core to our concurrent systems were invented alongside those distributed programming concepts.

> Hoare and Hansen monitors (1974).
>
> From semaphores to mutual exclusion: Dijkstra/Dekker (1965), Lamport (1974), Peterson (1981).
>
> Multithreading (Lamport 1979).

Most of the pioneers of concurrency are also the pioneers of distributed.

Beyond synchronization: communication

Furthering the distributed bent was focus on communication. Modern synchronization is a special case that (ab)uses shared memory space.

> Message passing: communication is done by sending messages.
>
> CSPs and Actors: independent processes and state machines.
>
> Futures and Promises: dataflow-based asynchronous communication.

Memory interconnects are just a form of hardware message passing.

The focus on structured communication was lost along the way, but is now coming back (Akka, Thrift, Go CSPs, gRPC, …).

The parallel 80's

Explosion of research into parallelism, largely driven by AI research.

The Connection Machine: up to 65,536 processors.

From this was born a "supercomputing" culture that persists to this day.

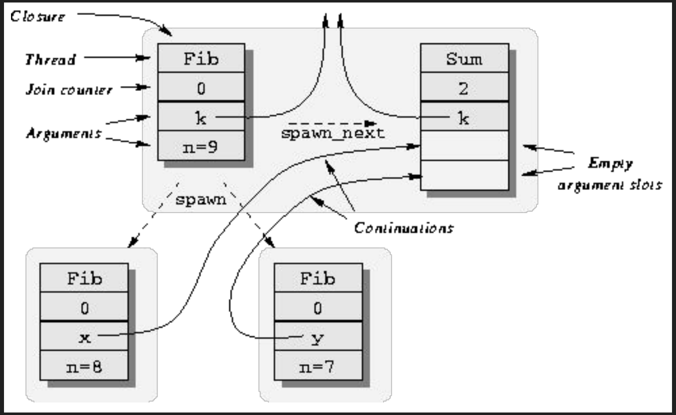
Parallel programming

Substantial focus on programmability.

- Task parallelism: computation drives parallel division.
- Data parallelism: data drives parallel division (scales with data).

Eventually, scheduling algorithms.

- Work stealing (Cilk).
- Nested parallelism (Cilk, NESL).



Parallel programming (Ex1. Connection Machine)

Programming 65,536 processors is different!

```
(DEFUN PATH-LENGTH (A B G)
    α(SETF (LABEL •G) +INF)
     (SETF (LABEL A) 0)
     (LOOP UNTIL (< (LABEL B) +INF)
         DO α(SETF (LABEL •(REMOVE A G))
             (1+ (βMIN α(LABEL •(NEIGHBORS •G))))))
     (LABEL B))
```

Every processor is a ~CSP.

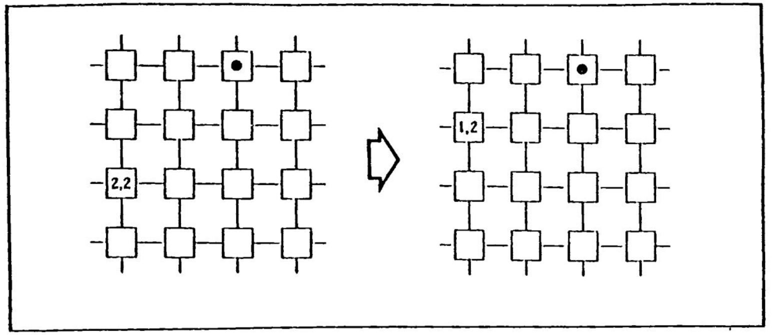


Figure 1.6: A simple (but inefficient) communications network topology

Parallel programming (Ex2. Cilk)

Bringing parallelism to imperative C-like languages.

```
// Task parallel:
cilk int fib(int n) {
    if (n < 2) { return n; }
    else {
        int x = spawn fib(n - 1);
        int y = spawn fib(n - 2);
        sync;
        return x + y;
    }
}
```

```
// Data parallel:
cilk_for (int i = 0; i < n; i++) {
    a[i] = f(a[i]);
}

cilk::reducer_opadd<int> result(0);
cilk_for (int i = 0; i < n; i++) {
    result += foo(i);
}
```

Foreshadows modern data parallel, including GPGPU and MapReduce.

Parallel programming (Ex3. OpenMP)

Eventual mainstream language adoption of these ideas (1997).

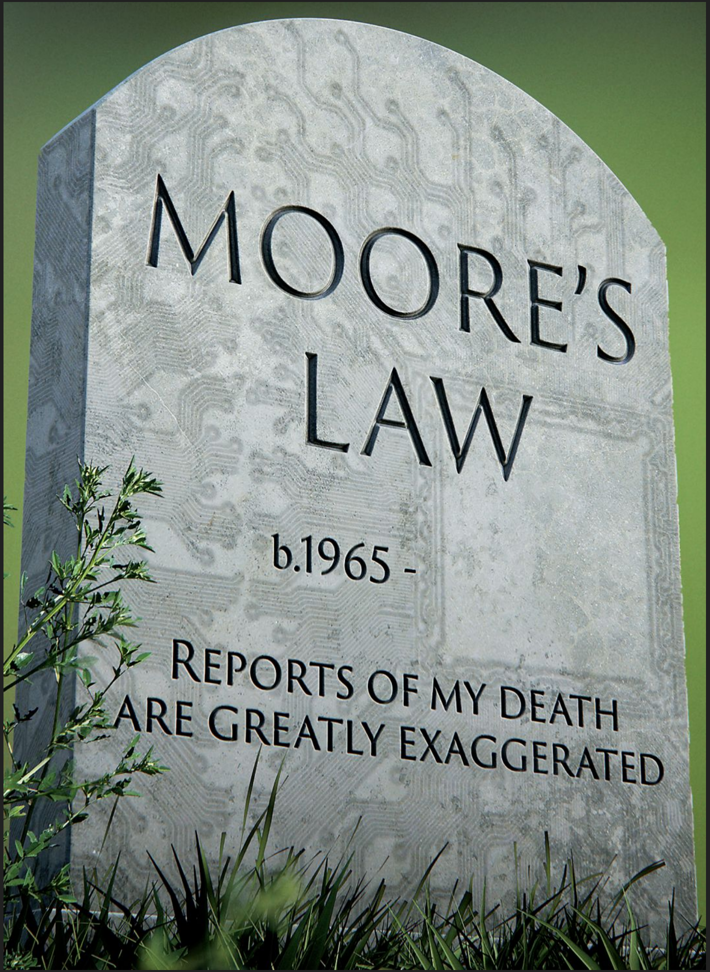
```
// Task parallel:
cilk int fib(int n) {
    if (n < 2) { return n; }
    else {
        #pragma omp task shared(n)
        int x = spawn fib(n - 1);
        #pragma omp task shared(n)
        int y = spawn fib(n - 2);
        #pragma omp taskwait
        return x + y;
    }
}
```

```
// Data parallel:
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    a[i] = f(a[i]);
}

#pragma omp for reduction(+: result)
for (int i = 0; i < n; i++) {
    result += foo(i);
}
```

Notice similarity to Cilk; set the stage for the future (Java, .NET, CUDA, ...).

Enter multi-core



Copyright © 2017 ACM

Moore’s Law: what happened?

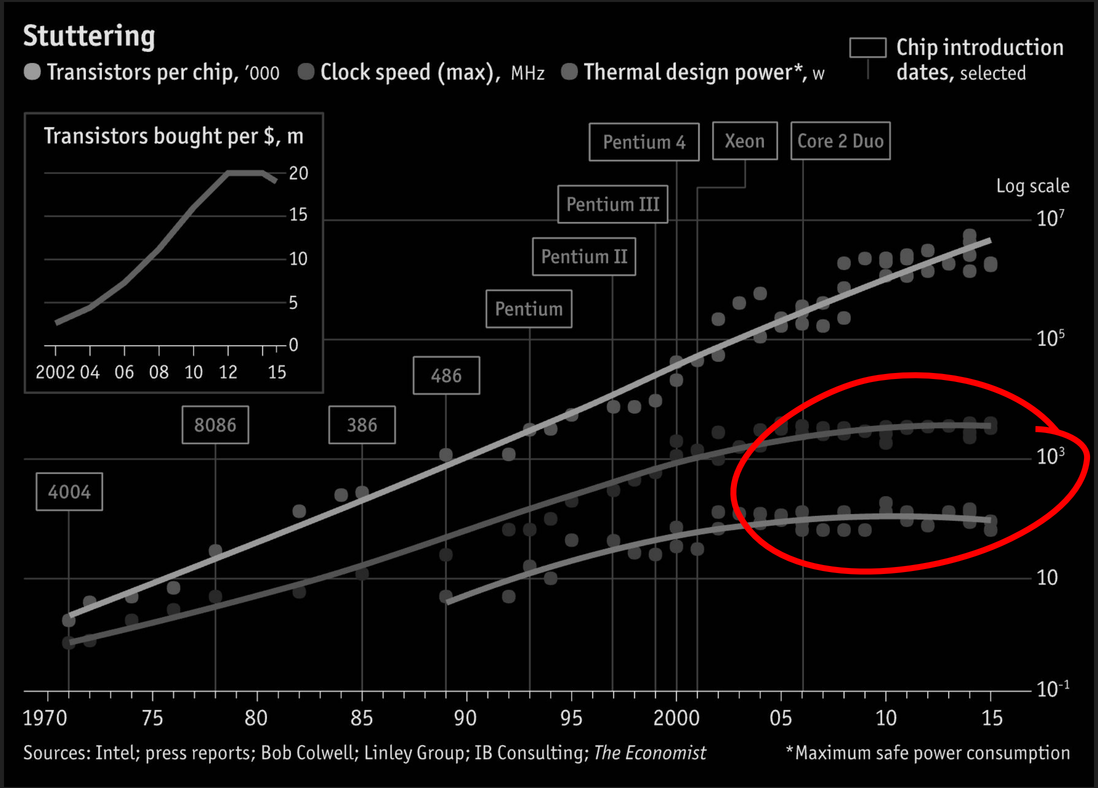
Clock speed doubled every ~18mos until 2005:

> *“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, ... there is no reason to believe it will not remain nearly constant for at least 10 years.”*
>
> *-- Gordon Moore, 1965*
>
> *"It can't continue forever. The nature of exponentials is that you push them out and eventually disaster happens.”*
>
> *-- Gordon Moore, 2005*

Power wall: higher frequency requires quadratic voltage/power increase, yielding greater leakage and inability to cool, requires new innovation.

Result: you don’t get faster processors, you just get more of them.

Moore’s Law: stuttering



The renaissance: we must program all those cores!

The 2000's was the concurrent programming renaissance.

Java, .NET, C++ went from threads, thread pools, locks, and events

to tasks, work stealing, and structured concurrency.

GPGPU became a thing, fueled by CUDA and architecture co-design.

Borrowed the best, easiest to program, ideas from academia. These building blocks now used as the foundation for asynchrony.

Somehow lost focus on CSPs, Actors, etc.; Erlang and Go got this right.

Modest progress on safety

Recognition of the importance of immutable data structures.

Short-lived industry-wide infatuation with software transactional memory.

- Some success stories: Haskell, Clojure, Intel TSX instructions.
- But overall, the wrong solution to the wrong problem.

Shared-nothing programming in some domains (Erlang), but mostly not.

A common theme: safety remains (mostly) elusive to this day.

Why did it matter, anyway?

Servers already used concurrency to serve many users.

- But “typical” client code did not enjoy this (no parallelism).
- It still largely doesn’t. Amdahl’s Law is cruel (1/((1-P)+(P/N))).

The beginning of the end of the Wintel Era:

- No more *“Andy giveth and Bill taketh away”* virtuous cycle

From this was born mobile, cloud computing, and GPGPU-style “supercomputer on every desk”. Less PC, but multi-core still matters.

Post multi-core era

The return of distributed programming

We have come full circle:

Distributed ⇒ concurrency ⇒ distributed.

Thanks to cloud, proliferation of devices (mobile + IoT).

Increasingly fine-grained distributed systems (microservices) look more and more like concurrent systems. We learned a lot about building these.

How much applies to our distributed future?

Thankfully, a lot. Let’s look at 7 key lessons.

1: Think about communication first, and always

Ad-hoc communication leads to reliability and state management woes.

CSP, actors, RPC, queues are good patterns. Stateless can help but is often difficult to achieve; most likely, ephemeral (recreatable).

Thinking about dependencies between services as “capabilities” can help add structure and security. Better than dynamic, weak discovery.

Queuing theory: imbalanced producers/consumers will cause resource consumption issues, stuttering, scaling issues. Think about flow control.

2: Schema helps; but don’t blindly trust it

Many concurrent systems use process-level “plugins” (COM, EJB, etc).

Schema is a religious topic. It helps in the same way static typing helps; if you find static typing useful, you will probably find schema useful.

But don’t blindly trust it. The Internet works here; copy it if possible.

The server revvs at a different rate than the client.

Trust but verify. Or, as Postel said:

> *“Be conservative in what you do, be liberal in what you accept.”*

3: Safety is important, but elusive

Safety comes in many forms: isolated » immutable » synchronized.

Lack of safety creates hazards: races, deadlocks, undefined behavior.

> Message passing suffers from races too; just not data races.
>
> State machine discipline can help, regardless of whether concurrent (multithreading) or distributed (CSPs, Actors, etc).
>
> Relaxed consistency can help, but is harder to reason about.

In a distributed system the “source of truth” is less clear (by design).

4: Design for failure

Things will fail; don’t fight it. Thankfully:

> *“The unavoidable price of reliability is simplicity.” (Tony Hoare)*

Always rendezvous and deal with failures intentionally.

Avoid “fire and forget” concurrency: implies mutable state and implicit dependencies. Complex failure modes. Think about parent/children.

Design for replication and restartability.

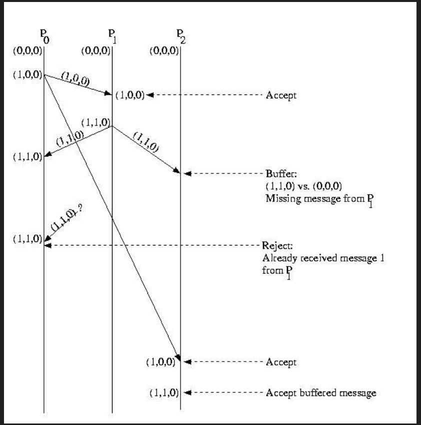
Error recovery is a requirement for a reliable concurrent system.

5: From causality follows structure

Causality is the cascade of events that leads to an action being taken. Complex in a concurrent system.

When context flows along causality boundaries, management becomes easier; but it isn't automatic:

- Cancellation and deadlines.
- Security, identity, secrets management.
- Tracing and logging.
- Performance profiling (critical path).

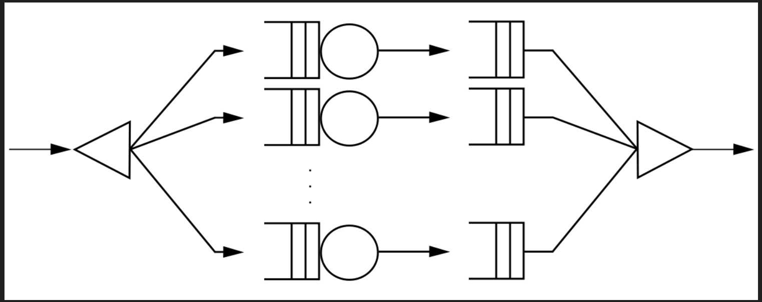


6: Encode structure using concurrency patterns

Easier to understand, debug, manage context, … everything, really.

A task without a consumer is a silent error or leak waiting to happen.

Structured concurrency instills structure: fork/join, pipeline. Let dataflow determine dependence for maximum performance (whether push or pull).



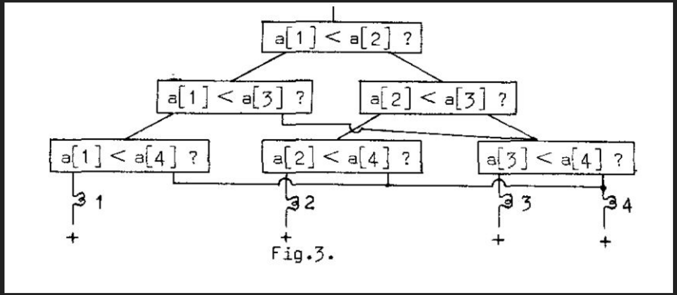


Fig.3.

6: Encode structure using concurrency patterns

For example, this

```
results := make(chan *Result, 8)
for _, e := range … {
    go crawl(url, results) // send to results as ready
}
var merged []*Result
for _, result := range results {
    merged = append(merged, process(result)) // do final processing
}
```

and not this (even though it is “easier”)

```
for _, url := range pages {
    go crawl(url) // fire and forget; crawler silently mutates some data structures
}
```

7: Say less, declare/react more

Declarative and reactive patterns delegate hard problems to compilers, runtimes, and frameworks: data locality, work division, synchronization.

```
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
```

Reactive models the event-oriented nature of an inherently asynchronous system. It is a declarative specification without impedance mismatch.

```
var keyups = Rx.Observable.fromEvent($input, 'keyup')
  .pluck('target', 'value').filter(text => text.length > 2);
```

Push is often the best for performance, but the hardest to program.

Serverless is a specialization of this idea (single event / single action).

Recap

1. Think about communication first, and always.
2. Schema helps; but don’t blindly trust it.
3. Safety is important, but elusive.
4. Design for failure.
5. From causality follows structure.
6. Encode structure using concurrency patterns.
7. Say less, declare/react more.

Future programming models will increasingly make these “built-in.”

In conclusion

Our concurrent past; our distributed future

Many design philosophies shared. Hopefully more in the future.

Expect to see more inspiration from the early distributed programming pioneers. Shift from low-level cloud infrastructure to higher-level models.

Reading old papers seems “academic” but can lead to insights.

> *“Those who cannot remember the past are condemned to repeat it.”*

Programming clouds of machines will become as easy as multi-core.

Thank you

Joe Duffy <joe@pulumi.com>

http://joeduffyblog.com

https://twitter.com/xjoeduffyx