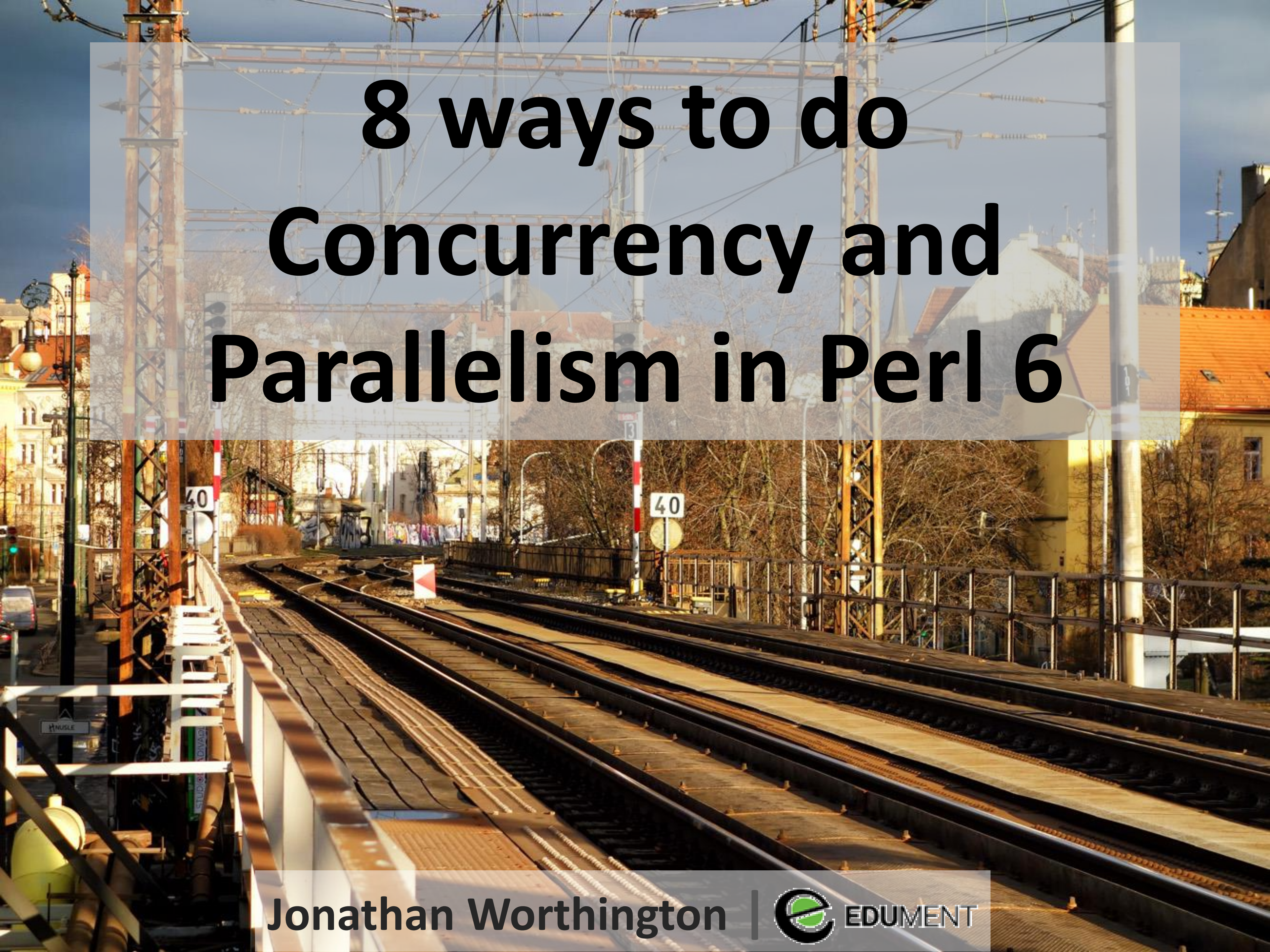


8 ways to do Concurrency and Parallelism in Perl 6

Jonathan Worthington | EDUMENT

Parallelism

Doing multiple things at the same time, in order to decrease wallclock time.

Part of the solution domain.

Concurrency

Multiple ongoing operations with overlapping start/end times.

Often part of the problem domain.

A parallel solution to a problem is correct if it produces *equivalent* results to a serial solution

but

Correctness is usually *far harder* to define in a concurrent system, and is as much a requirements issue as an implementation issue

Different problems require different tools to solve them

This session surveys various parallel and concurrent programming features on offer in Perl 6, both in core and in its modules, and looks at what problems they apply to

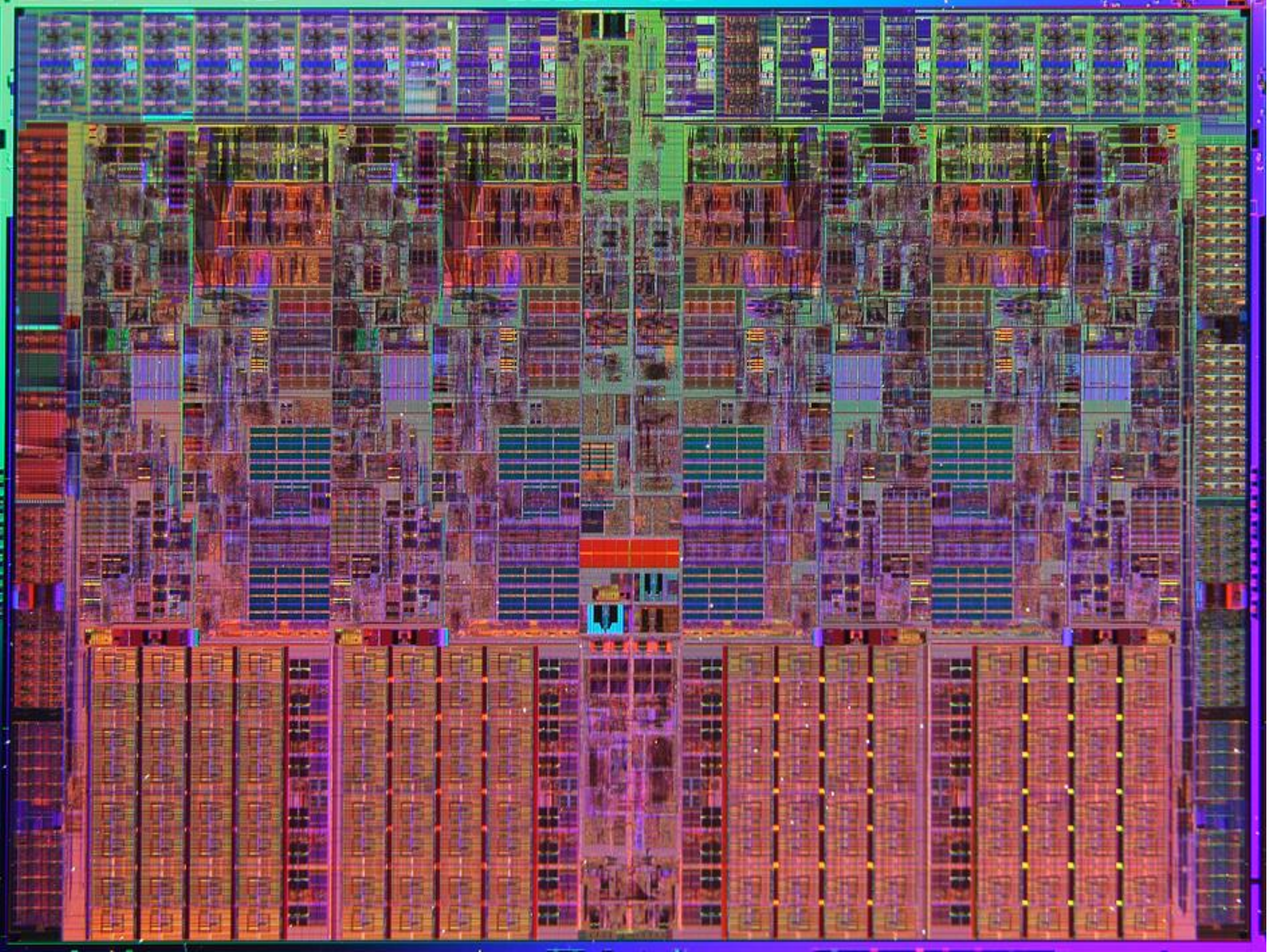

Threads, Mutexes,
Condition Variables,
Semaphores, etc.

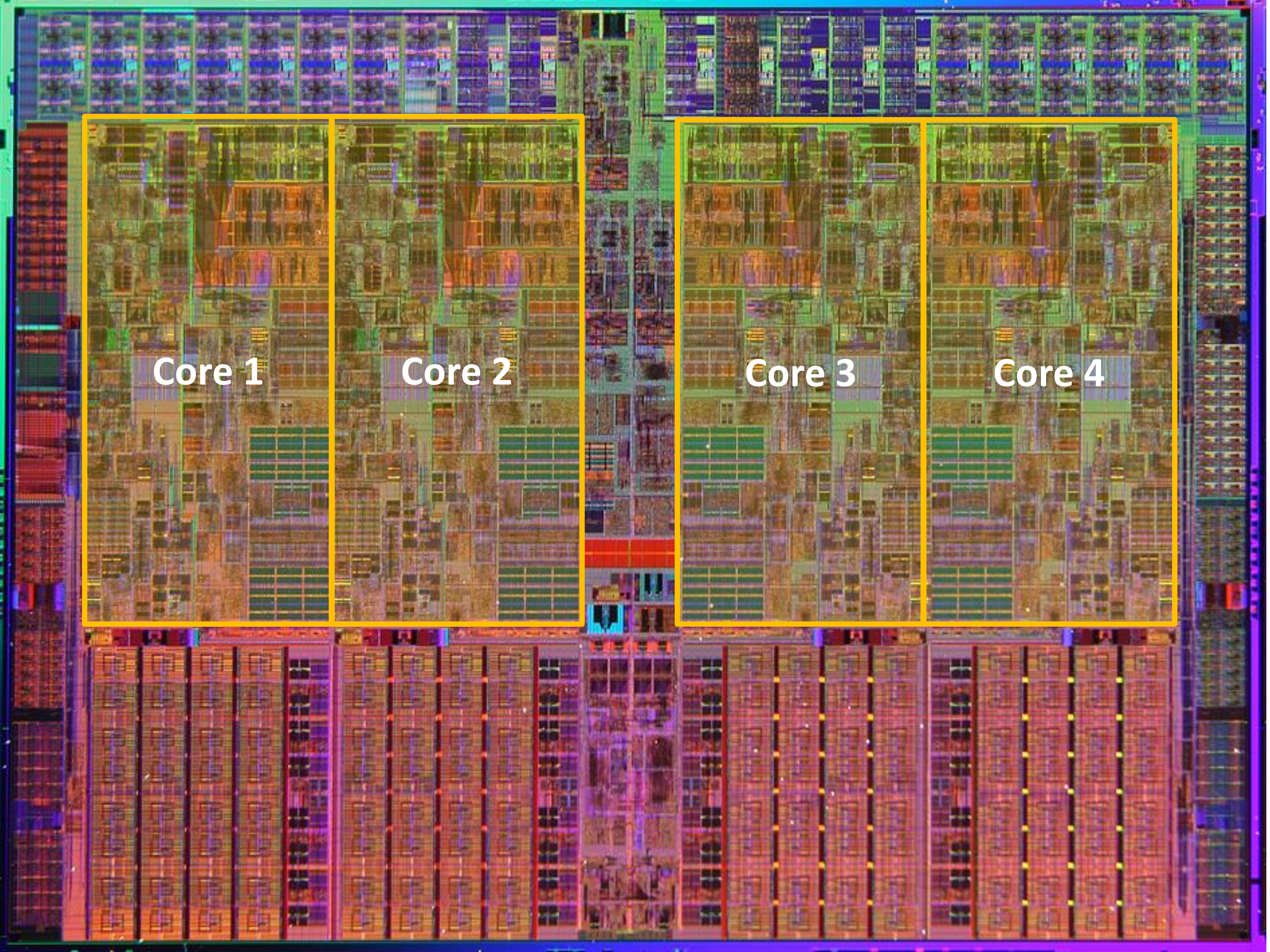
The "assembly language" of concurrency and parallelism

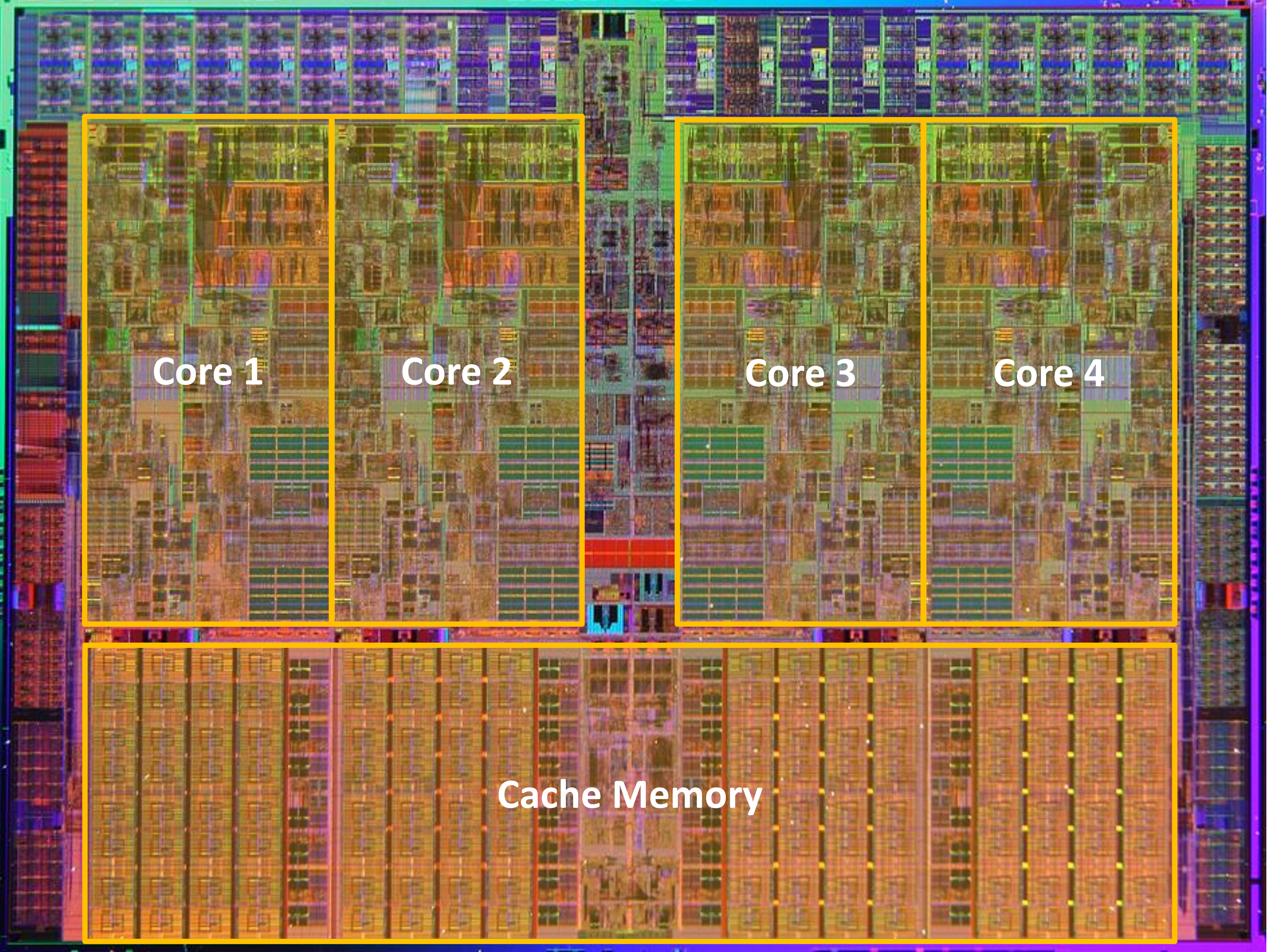
They make the hard things possible

and

The things that make the easy things easy are built on top of them




Core 1
Core 2
Core 3
Core 4


Core 1
Core 2
Core 3
Core 4
Cache Memory

A thread is scheduled on a core

Provided in Perl 6 by the Thread class

```
my @threads = do for 1..5 -> $id {
    Thread.start: {
        say "Hi from thread $id";
        sleep 1;
        say "Bye from thread $id"
    }
}
.join for @threads;
```

(Nearly) nothing is atomic

What will the output of this be?

```
my int $i = 0;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $i++ for ^100000;
    }
}
.join for @threads;
say $i;
```

Always remember:

There is no promise of execution ordering between threads, except that which you explicitly arrange for

Nothing a thread does is atomic or uninterruptible unless you explicitly arrange for it to be

The Lock class

A *reentrant* lock (that is, a given thread can lock/unlock it recursively)

Kernel supported, meaning the OS knows not to schedule a thread waiting for a lock until the lock is available

Correct answer, no parallel work

```
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.lock();
        $i++ for ^10000000;
        $lock.unlock();
    }
}
.join for @threads;
say $i;
```

Correct answer, no parallel work

```
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.lock();
        $i++ for ^10000000;
        $lock.unlock();
    }
}
.join for @threads;
say $i;
```

But never write code like this! Why?

Use protect to release the lock, even if an exception occurs

```
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $lock.protect: {
            $i++ for ^10000000;
        }
    }
}
.join for @threads;
say $i;
```

Parallel work, loads of contention

```
my int $i = 0;
my $lock = Lock.new;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        for ^10000000 {
            $lock.protect: { $i++ };
        }
    }
}
.join for @threads;
say $i;
```

Multiple threads trying to update the same data will perform *poorly*

To update data, the CPU core has to get it exclusively in its cache (so all other cores lose it from their cache)

60+ cycle penalty to get it back again!

And remember, locks are data too!

Other problems

A thread is not cheap to start/end

➔ Not ideal for fine-grained parallelism

No way to convey a result or failure

➔ But we almost always need to do so

"How many threads" is hard to answer

➔ Nice to have some good defaults

When to use Thread, Lock, etc.

When you need that level of control (for example, writing native bindings)

When you're implementing higher-level parallel/concurrent abstractions

These are not common situations!

Tasks on a Thread Pool

What is a thread pool?

One or more threads

+

A work queueing mechanism

The runtime decides how many threads are required, and can re-use them for different pieces of work over time

Minimal, boring example

```
for 1..10 -> $i {
    $*SCHEDULER.cue: {
        say "Task $i starting";
        sleep 0.5;
        say "Task $i done"
    }
}

sleep;
```

Fire and forget? *Really?*

We nearly always care about...

Getting the result of some work

or

Waiting until it's completed

and

Dealing with any errors

Introducing `Promise`

A synchronization construct that may be in one of three states:

Planned: operation planned/in progress
Kept: operation completed
Broken: operation failed

The start statement prefix

Schedules work on the thread pool and returns a Promise representing it

```
my ($input-config, $app-config) = await
    start {
        load-yaml slurp $input-file
    },
    start {
        from-json $_ with slurp $*HOME.add('.fooconf')
    }
```

The await subroutine

Waits for one or more Promise to be kept, returns a list of the results

```
my ($input-config, $app-config) = await
    start {
        load-yaml slurp $input-file
    },
    start {
        from-json $_ with slurp $*HOME.add('.fooconf')
    }
```

What is this good for?

Simple bits of task parallelism - that is to say, situations where we have two or more *different* tasks to set off in one go

Setting off work in the background that we will need later on

Dependent Tasks, Divide and Conquer

It is also possible to `await` inside of work running on the thread pool

This leads to an implicit *dependency graph* of work to be done

Especially suited to divide and conquer, where we recursively break down a problem into smaller pieces

A sequential merge sort

```
sub merge-sort(@values, $from = 0, $elems = @values.elems) {
    if $elems > 1 {
        my $divide = ($elems / 2).ceiling;
        merge
            merge-sort(@values, $from, $divide),
            merge-sort(@values, $from + $divide, $elems - $divide)
    }
    elsif $elems == 1 {
        (@values[$from],)
    }
    else {
        Empty
    }
}
```

A parallel merge sort

```
sub parallel-merge-sort(@values, $from = 0,
                        $elems = @values.elems) {
    if $elems > 500 {
        my $divide = ($elems / 2).ceiling;
        my ($left, $right) = await
            (start parallel-merge-sort(@values, $from, $divide)),
            (start parallel-merge-sort(@values, $from + $divide,
                                       $elems - $divide));
        merge $left, $right
    }
    else {
        merge-sort @values, $from, $elems
    }
}
```

Perl 6.c vs. Perl 6.d

In Perl 6.c, this spawns a load of threads. If there's really a lot of elements, it could reach the thread pool's upper limit.

In Perl 6.d, it spawns threads up to the number of CPU cores. No risk of deadlocking due to running out.

What's changed in Perl 6.d?

An `await` on a thread pool worker thread takes a continuation

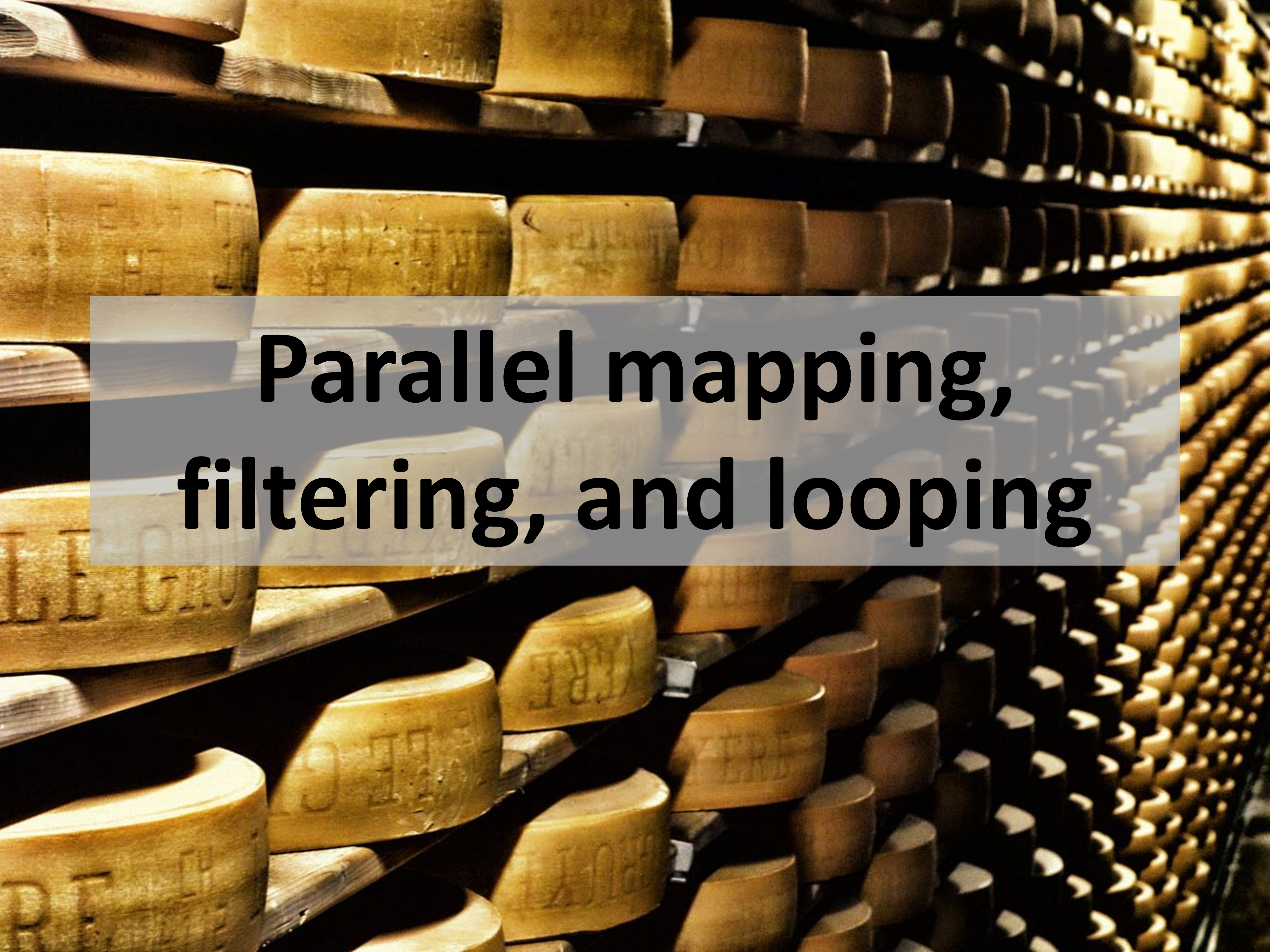
Schedules it to be resumed - quite possibly on a different pool thread - once the result is available

The pleasure of `await`
without the pain of `async`

When to use this approach

When a problem breaks down into parts that depend on each other, some of which can be done in parallel

(Many asynchronous operations are also return a `Promise`. The pattern works well for these also.)



Parallel mapping, filtering, and looping

Data parallelism

When we want to perform the *same operation* on many data items

Work may be compute bound or I/O bound (the latter will scale far better if using asynchronous I/O)

Parallel prime grep

Sequential runs in 17.2s

```
say ^100000 .grep(*.is-prime) .elems
```

Parallel runs in 5.3s

```
say ^100000 .race .grep(*.is-prime) .elems
```

hyper vs race

To preserve order of results relative to order of inputs, use hyper

If that doesn't matter, use race (you can get the first result faster, and there's less bookkeeping to do internally)

degree

How many parallel workers

(We try to pick a default based on the hardware. But you might want to use less resources, or know that your problem is I/O bound, not CPU bound.)

`batch`

The number of data items to give to a worker at a time

(You'll often want to tune this, based on knowledge of work per item and how important latency is. Lower values give better latency. Higher values give better throughput.)

Tweaked parallel prime grep

Default parallel runs in 5.3s...

```
say ^100000 .race .grep(*.is-prime) .elems
```

...but tweaking gets it to 4.1s*

```
say ^100000 .race(:1024batch, :12degree) .grep(*.is-prime) .elems
```

* On my 6-core workstation with hyper-threading enabled

A recent work example

We parse a file with various formulas, each of which we then parse/compile

```
method section:sym<output>($/) {
    make 'output' => [$<output>.map({
        my %props = .ast;
        with %props<formula> -> $formula {
            my $ast = parse-formula($formula);
            %props<compiled-formula> = compile-formula($ast);
        }
        Foo::Model::Output.new(|%output-props)
    })];
}
```

A recent work example

The work for each is independent, but order matters...

```
method section:sym<output>($/) {
    make 'output' => [$<output>.hyper.map({
        my %props = .ast;
        with %props<formula> -> $formula {
            my $ast = parse-formula($formula);
            %props<compiled-formula> = compile-formula($ast);
        }
        Foo::Model::Output.new(|%output-props)
    })];
}
```

A recent work example

...and there's few formulas, but quite a bit of work for each one

```
method section:sym<output>($/) {
    make 'output' => [$<output>.hyper(batch => 1).map({
        my %props = .ast;
        with %props<formula> -> $formula {
            my $ast = parse-formula($formula);
            %props<compiled-formula> = compile-formula($ast);
        }
        Foo::Model::Output.new(|%output-props)
    })];
}
```

When to use this approach

When you have the same work to do for a whole set of data items

When the work for each is *independent* from that of other data items (so there's no shared state needed between them)

Monitors

Objects and concurrency?

Objects are stateful, and state makes concurrency hard

but

OO correctly applied bounds access to mutable state to the object's methods

Tell, don't ask

Good OO designs have very few getters and query methods

Instead, they are heavy on command methods - that is, we send objects messages telling them what to do

Follow this design rule, and the object boundary is a natural concurrency control boundary

```
class Index {
    has $!lock = Lock.new;
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        $!lock.protect: { ... }
    }

    method append-docs(Str $word, @target --> Nil) {
        $!lock.protect: { ... }
    }

    method elems(--> Int) {
        $!lock.protect: { ... }
    }
}
```

```
class Index {
    has $!lock = Lock.new;
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        $!lock.protect: { ... }
    }

    method append-docs(Str $word, @target --> Nil) {
        $!lock.protect: { ... }
    }

    method elems(--> Int) {
        $!lock.protect: { ... }
    }
}
```

Repetitive!
Tedious!
Easy to forget!

OO::Monitors

Uses meta-programming to insert the locking around methods automatically

(Also supports conditions variables, for more advanced use cases)

```
use OO::Monitors;

monitor Index {
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        %!index{$word}{$document} = True;
    }

    method append-docs(Str $word, @target --> Nil) {
        @target.append(.keys) with %!index{$word};
     }

    method elems() {
        %!index.elems
    }
}
```

```
use OO::Monitors;

monitor Index {
    has %!index{Str};

    method add(Str $word, Str $document --> Nil) {
        %!index{$word}{$document} = True;
    }

    method append-docs(Str $word, @target --> Nil) {
        @target.append(.keys) with %!index{$word};
     }

    method elems() {
        %!index.elems
    }
}
```

Pass in array to append to ➔ avoids a query method and risk of laziness bug

When to use this approach

When you have state that needs to be used concurrently, and there's no other built-in mechanism that can provide that

Onus is still very much on the developer to do a good OO design

Lock-free Data Structures

What does lock-free mean?

A data structure that you can use concurrently without the need for locks

Not just that your code doesn't need locks, but also that the data structure itself doesn't use locks internally

How is this possible?!

How is this possible?!

CPUs provide atomic operations.
Perl 6 provides access to them.

```
my atomicint $i = 0;
my @threads = do for 1..5 -> $id {
    Thread.start: {
        $i⚛++ for ^100000;
    }
}
.join for @threads;
say $i;
```

Atomic Increment Operator

Atomic increment and atomic addition can sometimes be handy

Far more powerful is the atomic compare and swap operation, commonly known as "CAS"

CAS is provided by the hardware, but we can imagine it like this - with the guarantee that it is atomic

```
sub cas($reference is rw, $expected, $new) {
    my $seen = $reference;
    $reference = $new if $seen =:= $expected;
    return $seen;
}
```

Amazingly, we can make any data structure we want atomically updateable using CAS.*

*** If we follow the rules. Very, very carefully. Efficiency will vary widely by data structure.**

As an example, let's implement a lock-free stack data structure

Supports concurrent pushes and pops

```
class ConcurrentStack {
    ...
}
```

It's a linked list of Node objects. They nodes themselves are immutable. The only mutable thing will be `$!head`.

```
class ConcurrentStack {
    my class Node {
        has $.value;
        has Node $.next;
    }
    has Node $!head;

    method push($value --> Nil) { ... }

    method pop() { ... }
}
```

Here is push. This retry loop structure is typical of lock-free algorithms. If we must retry, it's because another thread succeeded ➔ global progress bound

```
method push($value --> Nil) {
    loop {
        my $next = $!head;
        my $new = Node.new: :$value, :$next;
        last if cas($!head, $next, $new) === $next;
    }
}
```

The pop method is similar, except it can fail due to an empty stack

```
method pop() {
    loop {
        my $cur = $!head;
        fail "Stack is empty" without $cur;
        if cas($!head, $cur, $cur.next) === $cur {
            return $cur.value;
        }
    }
}
```

This retry loop structure is so common, Perl 6 provides a form of CAS that takes a block computing the new value based on the current one, and does the retry loop automatically for us

```
method push($value --> Nil) {
    cas $!head, -> $next {
        Node.new: :$value, :$next
    }
}

method pop() {
    my $taken;
    cas $!head, -> $current {
        fail "Stack is empty" without $current;
        $taken = $current.value;
        $current.next
    }
    return $taken;
}
```

Modules available so far

Concurrent::Queue
Concurrent::Stack
Concurrent::Trie

When to use this approach

When the data structure you need has a lock-free implementation available

When you don't need blocking

(A lock-free queue would not be a good choice for a thread pool's work queue, because it must block efficiently when there is no work to do.)

Reactive Streams

Streams of asynchronous values

A `Promise` represents an asynchronous operation that produces a result

A `Supply` represents an asynchronous operation that produces many results over time (it may be finite or infinite)

Examples

Packets arriving over a socket
Output from a spawned process
GUI events
Ticks of a timer
Messages from a message queue
Domain events

Syntactic relief

Perl 6 provides syntactic support for working with asynchronous streams

At the heart of it are `react` and `supply` blocks, which enforce one-at-a-time message processing even when dealing with many data sources

An asynchronous web crawler

```
use Cro::HTTP::Client;

sub crawl($initial-url) {
    react {
        my %seen;
        my $client = Cro::HTTP::Client.new;
        crawl-url($initial-url);

        sub crawl-url($url) {
            ...
        }
    }
}
```

An asynchronous web crawler

```
sub crawl-url($url) {
    return if %seen{$url}++;
    say "Getting $url";
    whenever $client.get($url) -> $response {
        if $response.content-type.type-and-subtype
                eq 'text/html' {
            get-links($response, $url);
        }
        QUIT {
            default {
                note "$url failed: " ~ .message;
            }
        }
    }
}
```

An asynchronous web crawler

```
sub get-links($response, $base) {
    whenever $response.body-text -> $text {
        for $text.match(/'href="' <!before \w+':'>
                <( <-["]>+/, :g) {
            crawl-url cat-uri $base, ~$_;
        }
    }
}
```

What's being done for us?

Concurrency control, to protect our state (the %seen URL hash)

Tracking outstanding work, and terminating when there's no more

Propagating any errors we forget

When to use this approach

Whenever your problem looks like - or can be seen as - a stream of events

A *lot* of concurrent problems can be seen this way. Further, many concurrency tasks become clearer when considered as an event processing problem.

Channels and Workers

Introducing `Channel`

A blocking concurrent queue, which can also convey error and completion

Safe for multiple threads to `send` values

Safe for multiple threads to (compete to) `receive` values

`Channel` vs. `Supply`

With a `Supply`, the sender pays the costs of processing a message (thus providing a backpressure mechanism)

With a `Channel`, the receiver pays the cost of processing a message (plus there's a memory cost for the queue)

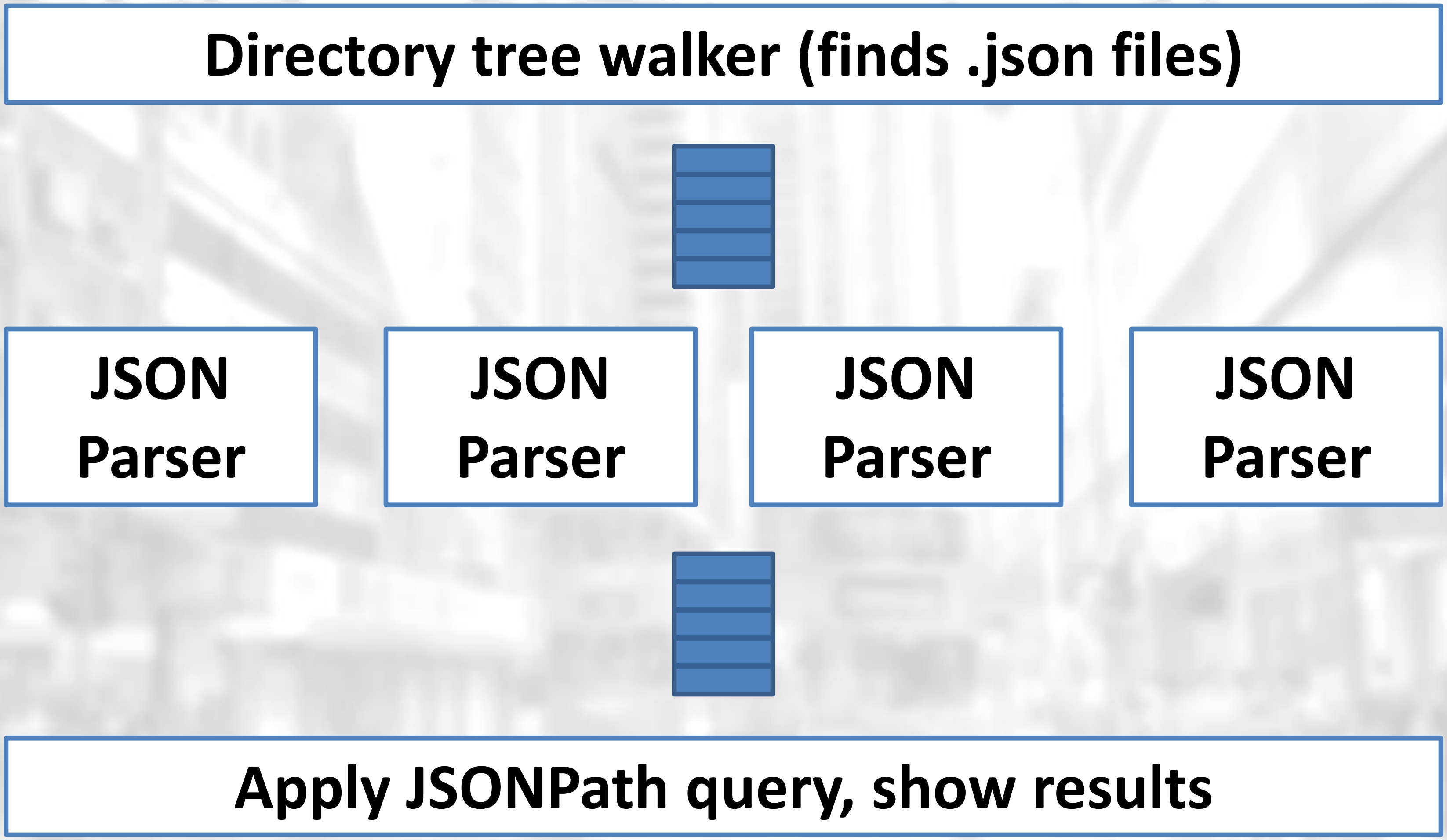
Staged Event-Driven Architecture

Build a system out of a set of stages that are joined together by `Channels`

For stages where it is safe to do so, can spawn multiple workers

Queue lengths show bottlenecks

Example: json-search



Example: json-search

Make channels and spawn workers

```
use JSON::Fast;
use JSON::Path;

sub MAIN(Str $query, Str $dir = '.') {
    my $to-parse = Channel.new;
    my $to-search = Channel.new;
    my $finder = start find-json-files($dir, $to-parse);
    my @parsers = (start parse $to-parse, $to-search) xx 8;
    Promise.allof(@parsers).then({ $to-search.close });
    my $searcher = start search $to-search, $query;
    await $finder, @parsers, $searcher;
}
```

Example: json-search

Look for JSON files, send the paths

```
sub find-json-files($start-dir, $to-parse) {
    sub walk($dir) {
        for dir($dir) {
            when .d { walk($_); }
            when .f && .extension eq 'json' {
                $to-parse.send($_);
            }
        }
    }
    walk($start-dir.IO);
    $to-parse.close;
}
```

Example: json-search

Parse each file, send on the result

```
sub parse($to-parse, $to-search) {
    for $to-parse.list -> $path {
        $to-search.send(SearchFile.new(
            :$path, :json(from-json(slurp($path)))));
        CATCH {
            default {
                note .message;
                $to-search.send(SearchFile.new(
                    :$path, :error(.message)));
            }
        }
    }
}
```

Example: json-search

Query the data and show results

```
sub search($to-search, $query) {
    my $path = JSON::Path.new($query);
    for $to-search.list {
        if .error {
            note "ERROR {.path}: {.error}";
        }
        orwith $path.value(.json) -> $result {
            say "{.path} &to-json($result)";
        }
    }
}
```

`whenever` and `Channel`

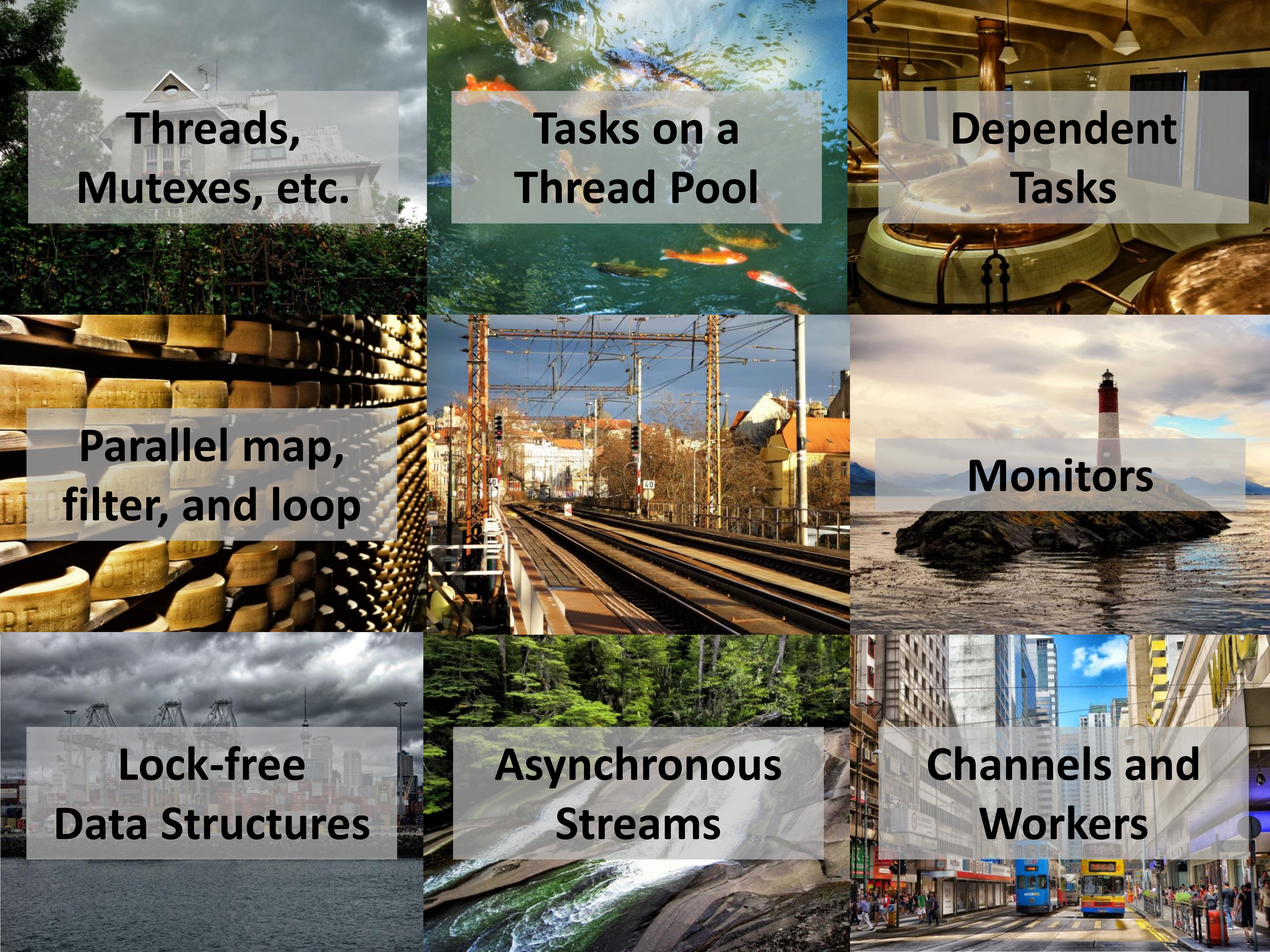
It's also possible to consume values from a `Channel` reactively

This allows multiplexing channels themselves, or even multiplexing channels with supplies and promises

When to use this approach

When you need "receiver pays" semantics for messages

When wanting to build work pipelines and dedicate a thread to each worker (or multiple for stateless workers)


Threads, Mutexes, etc.
Tasks on a Thread Pool
Dependent Tasks
Parallel map, filter, and loop
Monitors
Lock-free Data Structures
Asynchronous Streams
Channels and Workers


LE CRU


Thank you!
Questions?