

Parallelism, Concurrency, and Asynchrony in Perl 6

Jonathan Worthington

A man with a beard, wearing a black beanie and a dark jacket, stands on a vast, textured glacier. He is holding a trekking pole in his right hand. The background features steep, rocky mountains under a clear sky. The text "Hi! I'm Jonathan." is overlaid in a bold, dark blue font.

Hi!
I'm Jonathan.

A person with a beard, wearing a black beanie, a dark jacket with a 'FOX' logo, and a backpack, stands in a snowy mountain landscape. They are holding a ski pole. The background shows snow-covered mountains and a valley.

**Lead developer of
Rakudo Perl 6**

**Founder and architect
of MoarVM**

**Work as a software
architect and teacher**

Live in beautiful Prague

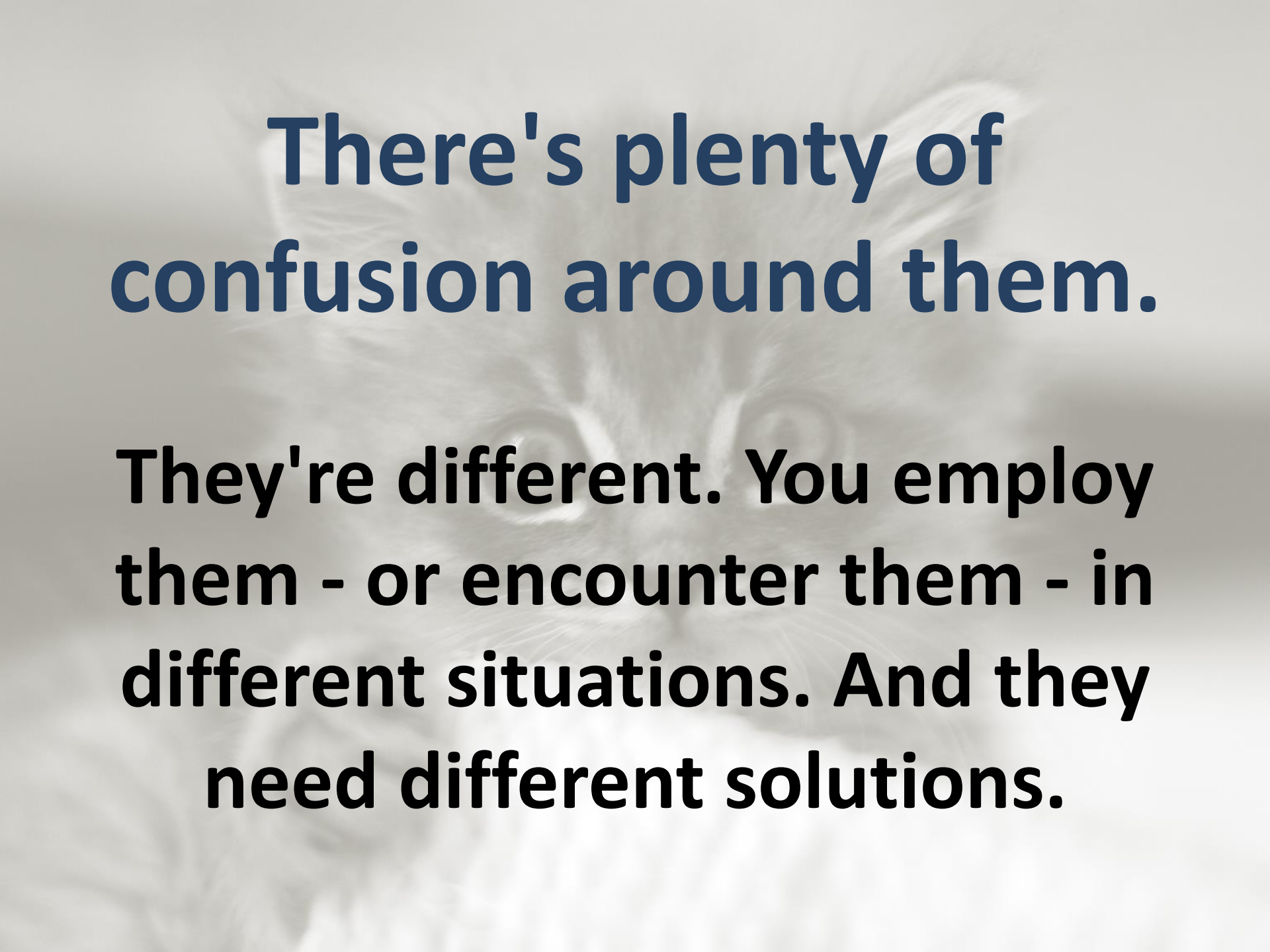
A blurred background image of a cat's face, looking directly at the camera with wide, light-colored eyes. The cat's fur is light-colored, and its whiskers are visible.

My 3 topics for today:

Parallelism

Asynchrony

Concurrency



**There's plenty of
confusion around them.**

**They're different. You employ
them - or encounter them - in
different situations. And they
need different solutions.**

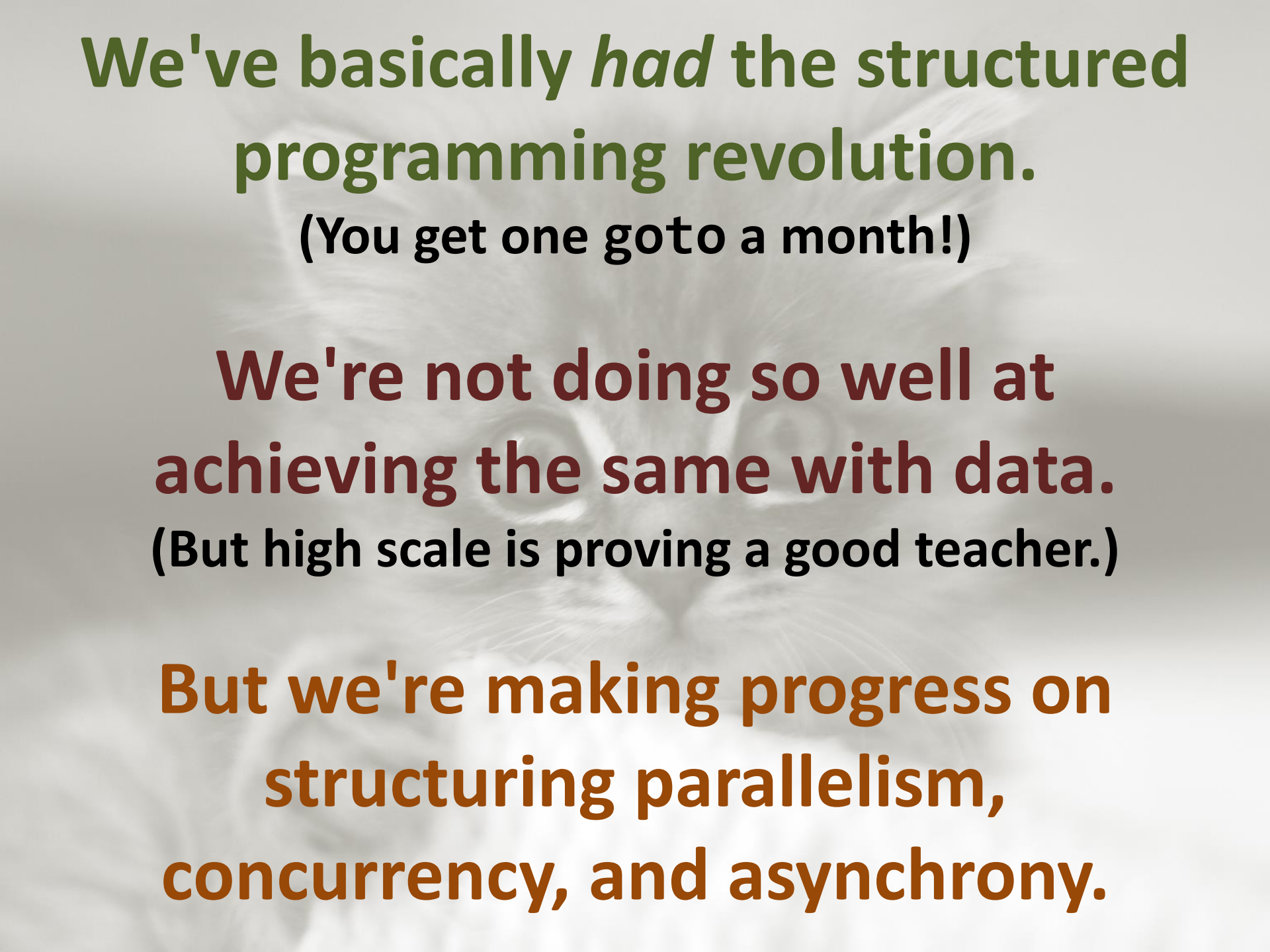


**These topics have a
history of pain.**

Threads (and data races)

Locks (and deadlocks)

**Condition variables
(and spurious wakeups)**



We've basically *had* the structured programming revolution.

(You get one goto a month!)

We're not doing so well at achieving the same with data.

(But high scale is proving a good teacher.)

But we're making progress on structuring parallelism, concurrency, and asynchrony.



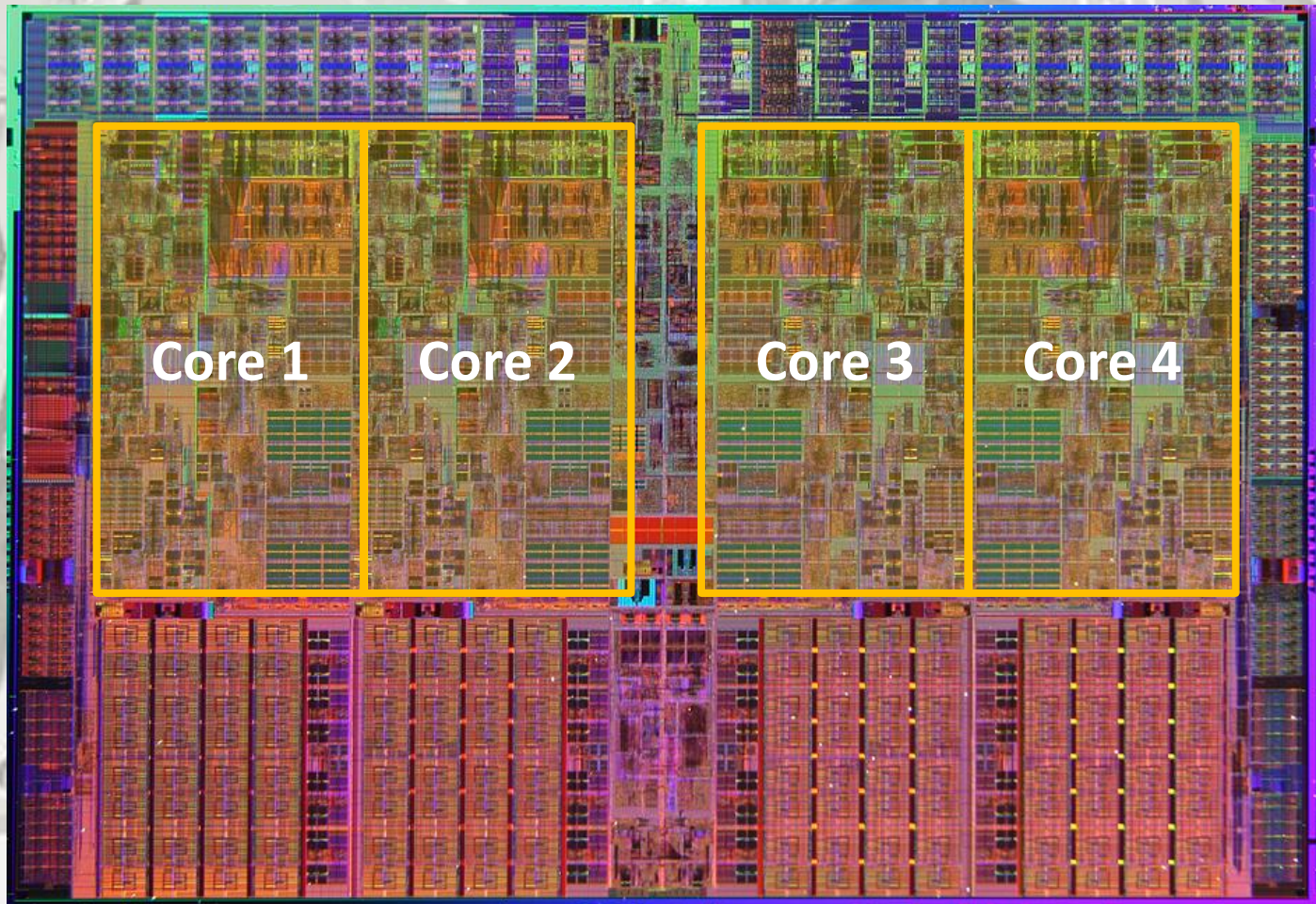
**I'm going to take you
through some of these
structured approaches.**

**And, since I've been working on
Perl 6 implementations of them,
I'll use that for my examples.**

A background image showing construction workers on a steel framework. One worker is in the foreground, wearing a hard hat and safety harness, working on a horizontal beam. Another worker is visible in the background, also on the framework. The image is semi-transparent, allowing the text to be overlaid.

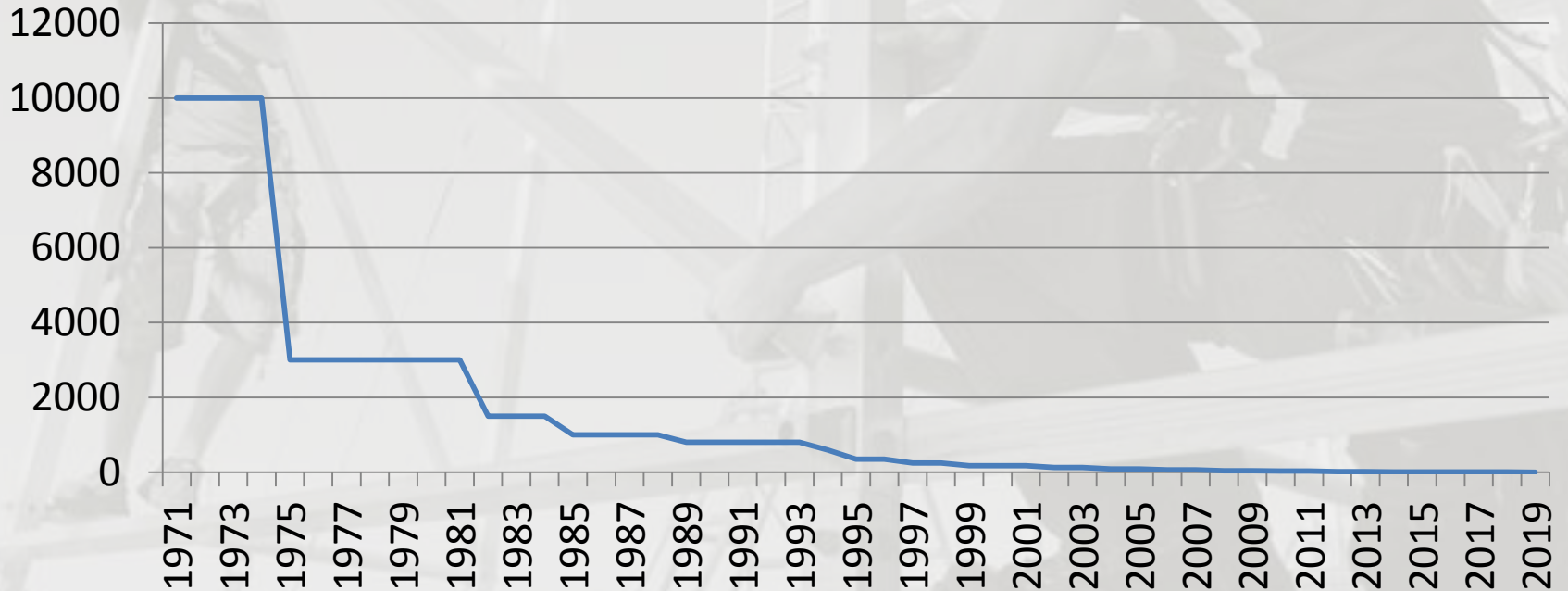
Let's start with
parallelism

Most modern CPUs have multiple cores



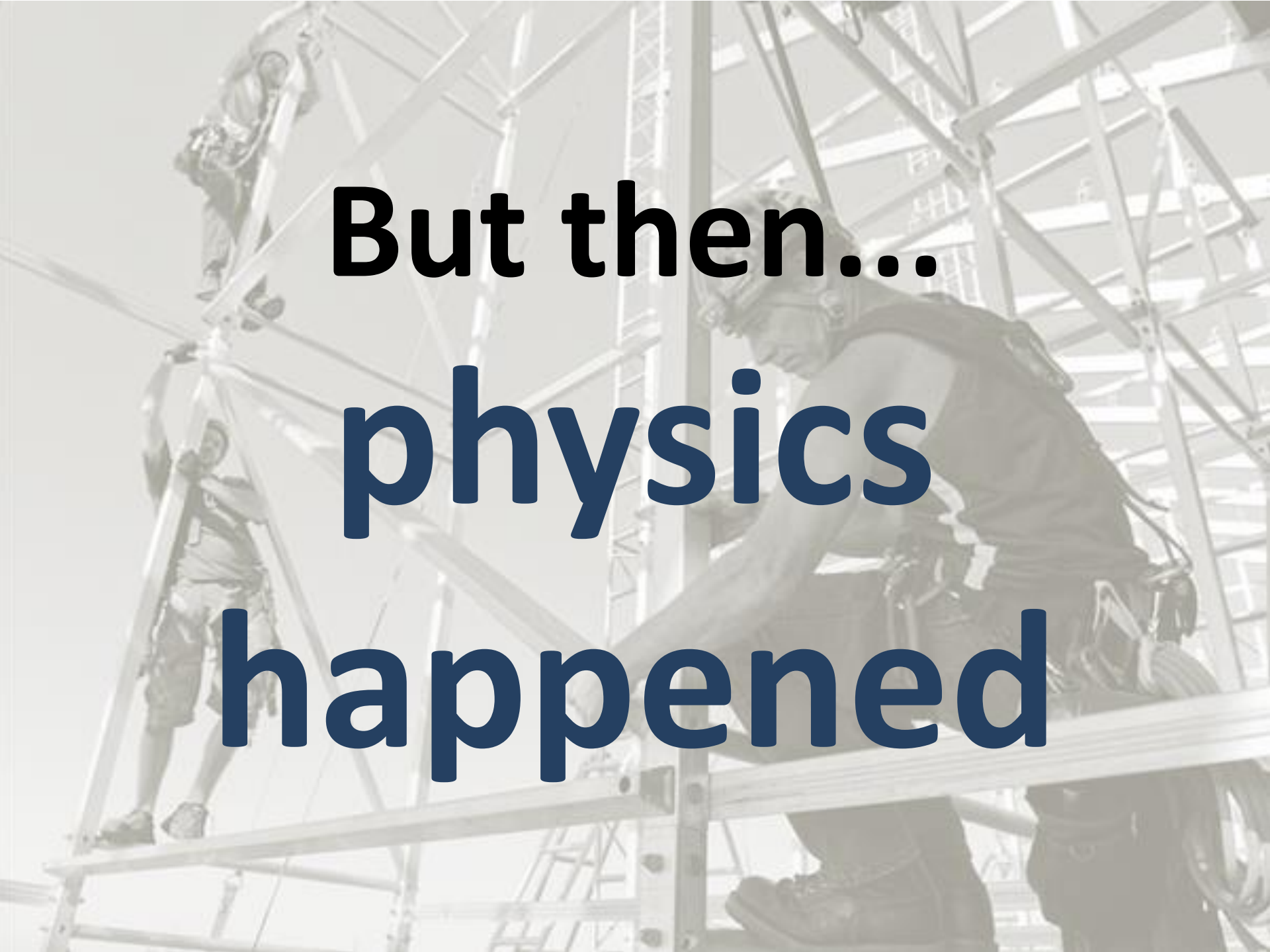
For years, we **increased CPU clock speeds by decreasing feature size,** bringing components closer so we could compute things faster

Feature sizes over time (nm)



But then...



A background image showing construction workers on a steel framework. One worker is in the foreground, leaning over a beam, while others are visible higher up on the structure. The image is semi-transparent, allowing the text to be clearly visible.

But then...
physics
happened



Speed of light – the
fastest we can
move information

$3.0 \times 10^8 \text{ m/s}$

$3.0 \times 10^9 \text{ Hz}$

Number of cycles a
3GHz CPU does per
second

$= 10 \text{ cm}$

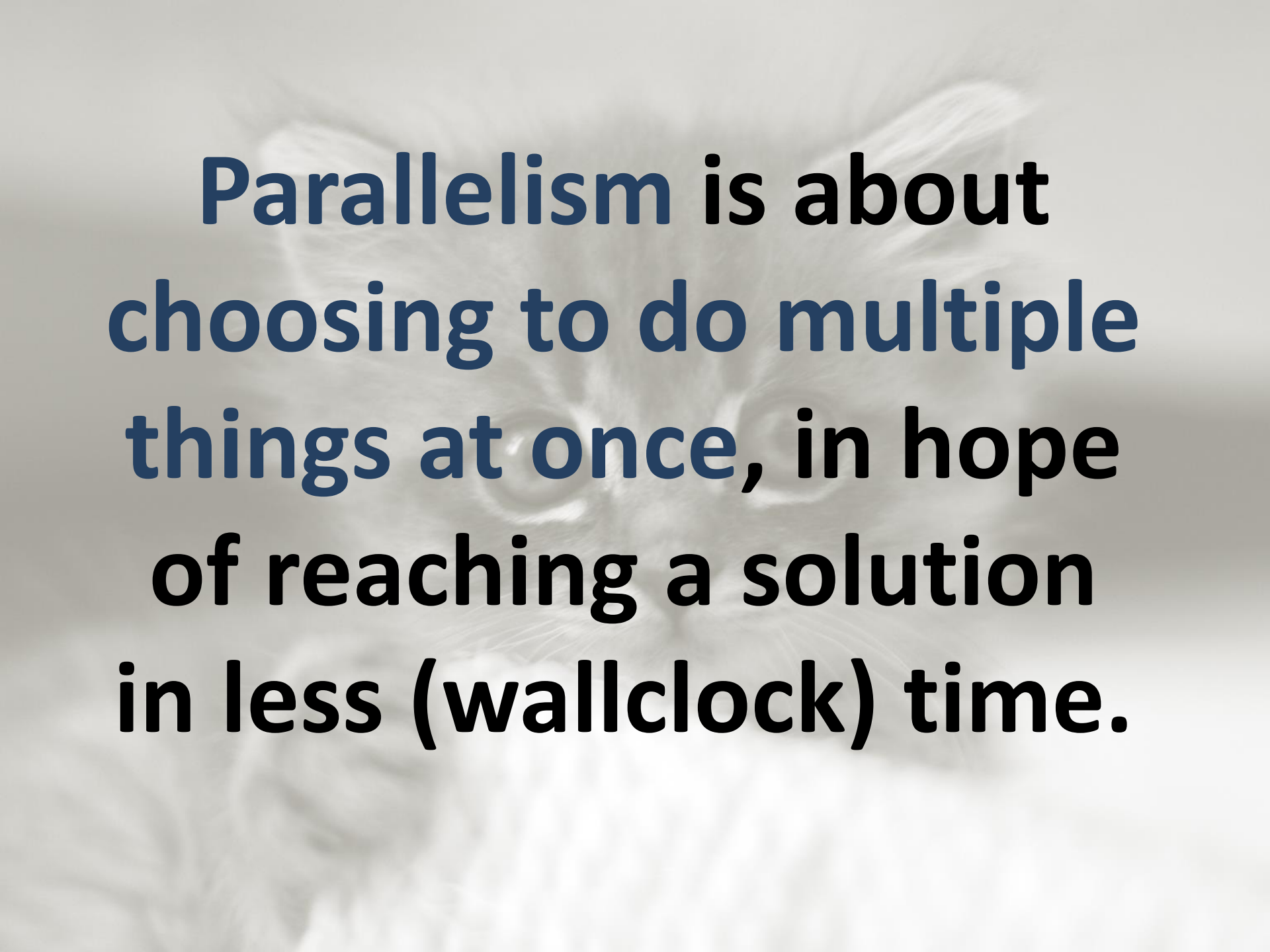
How far we can
move information
per cycle

A grayscale background image showing construction workers on a large steel framework, likely a bridge or industrial structure. The workers are positioned at different heights, with one worker in the foreground appearing to be working on a horizontal beam. The image is semi-transparent, allowing the text to be overlaid clearly.

**We're already
down to making
transistors out of
just 10s of atoms**

A background image showing three construction workers on a complex steel framework. One worker is in the foreground, leaning over a horizontal beam. Two other workers are visible in the background, one higher up and one lower down, both working on the structure. The image is semi-transparent, allowing the text to be clearly visible.

**The solution:
do multiple
things at the
same time**



**Parallelism is about
choosing to do multiple
things at once, in hope
of reaching a solution
in less (wallclock) time.**

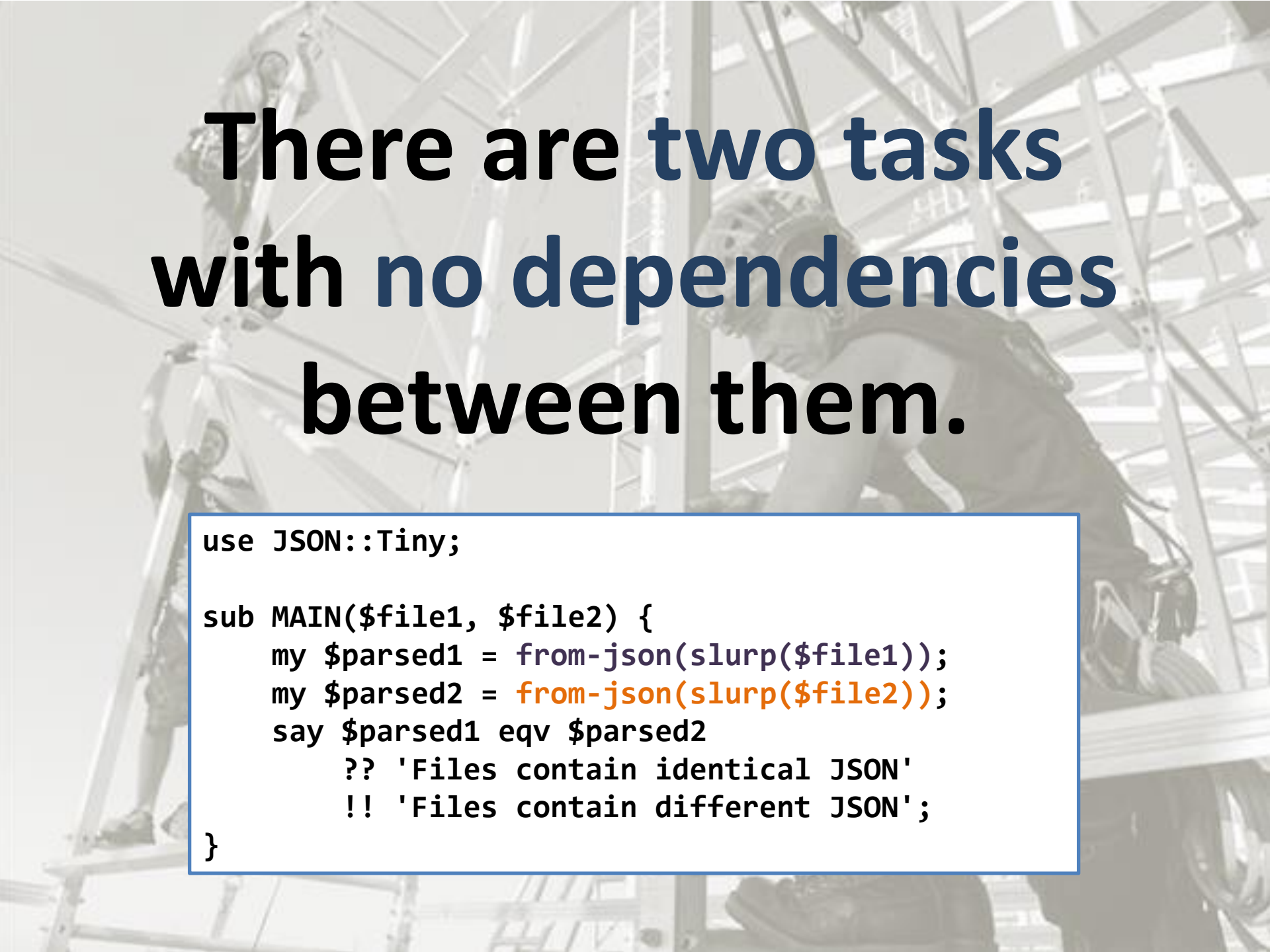
Parallelism is about
choosing to do multiple
things at once, in hope
of reaching a solution
in less (wallclock) time.

Parallelism is not
something that results
from the problem at
hand, but from our
choice of solution!

Here's a simple JSON comparison program.

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsed1 = from-json(slurp($file1));
    my $parsed2 = from-json(slurp($file2));
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```



There are two tasks with no dependencies between them.

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsed1 = from-json(slurp($file1));
    my $parsed2 = from-json(slurp($file2));
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```

A grayscale photograph of construction workers on a steel framework. One worker is in the foreground, leaning over a horizontal beam, while others are visible in the background at different levels of the structure. The image is semi-transparent, allowing the text to be overlaid.

**Let's do
them in
parallel!**

But...



The background of the slide is a grayscale photograph of construction workers on a large steel framework. One worker is visible in the upper left, another in the center right, and a third in the lower left. They are working on a complex network of steel beams and girders.

But...

How to pick the best number of tasks to work on at a time?

How to wait correctly and efficiently for completion, and correctly get the results?

How do we correctly handle exceptions in parallel work?

Use Promises!

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsing1 = start from-json(slurp($file1));
    my $parsing2 = start from-json(slurp($file2));
    my ($parsed1, $parsed2) = await $parsing1, $parsing2;
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```

Use Promises!

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsing1 = start from-json(slurp($file1));
    my $parsing2 = start from-json(slurp($file2));
    my ($parsed1, $parsed2) = await $parsing1, $parsing2;
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```

Schedule work on the thread pool. It can consider hardware, memory, etc. to choose appropriate number of workers.

Use Promises!

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsing1 = start from-json(slurp($file1));
    my $parsing2 = start from-json(slurp($file2));
    my ($parsed1, $parsed2) = await $parsing1, $parsing2;
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```

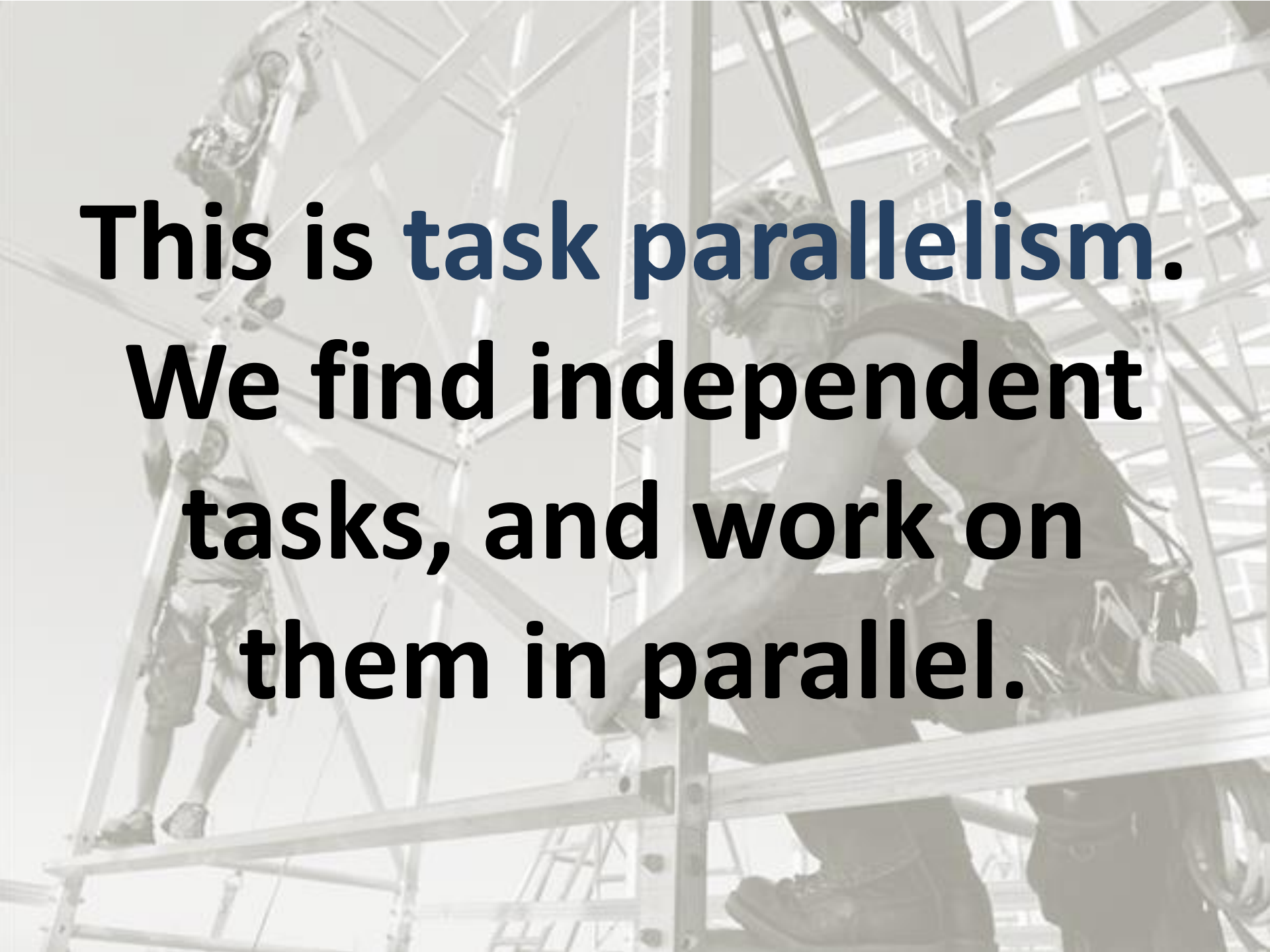
Can await any number of Promises; it waits efficiently (OS kernel aware) and unpacks the results for you

Use Promises!

```
use JSON::Tiny;

sub MAIN($file1, $file2) {
    my $parsing1 = start from-json(slurp($file1));
    my $parsing2 = start from-json(slurp($file2));
    my ($parsed1, $parsed2) = await $parsing1, $parsing2;
    say $parsed1 eqv $parsed2
        ?? 'Files contain identical JSON'
        !! 'Files contain different JSON';
}
```

Any exceptions thrown by the started code are captured, and then automatically re-thrown on the awaiting thread.

A grayscale background image showing three construction workers on a complex steel framework. One worker is in the foreground, leaning over a horizontal beam. Two other workers are visible in the background, one higher up and one lower down, both working on different parts of the structure. The image illustrates the concept of task parallelism in a construction context.

This is task parallelism.
We find independent
tasks, and work on
them in parallel.

Here's our next challenge:





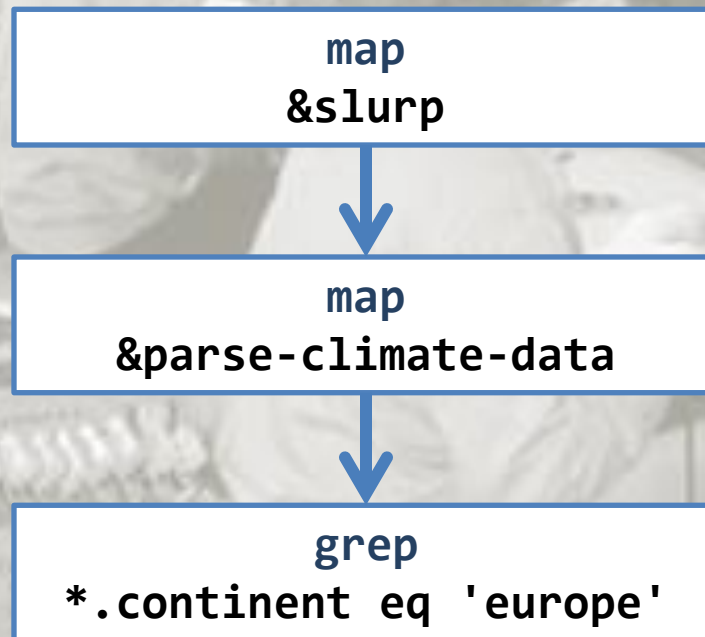
Here's our next challenge:
parallelizing a program that
parses many data files from
weather stations, filters out
those in Europe, and then
finds the place with the
maximum temperature.

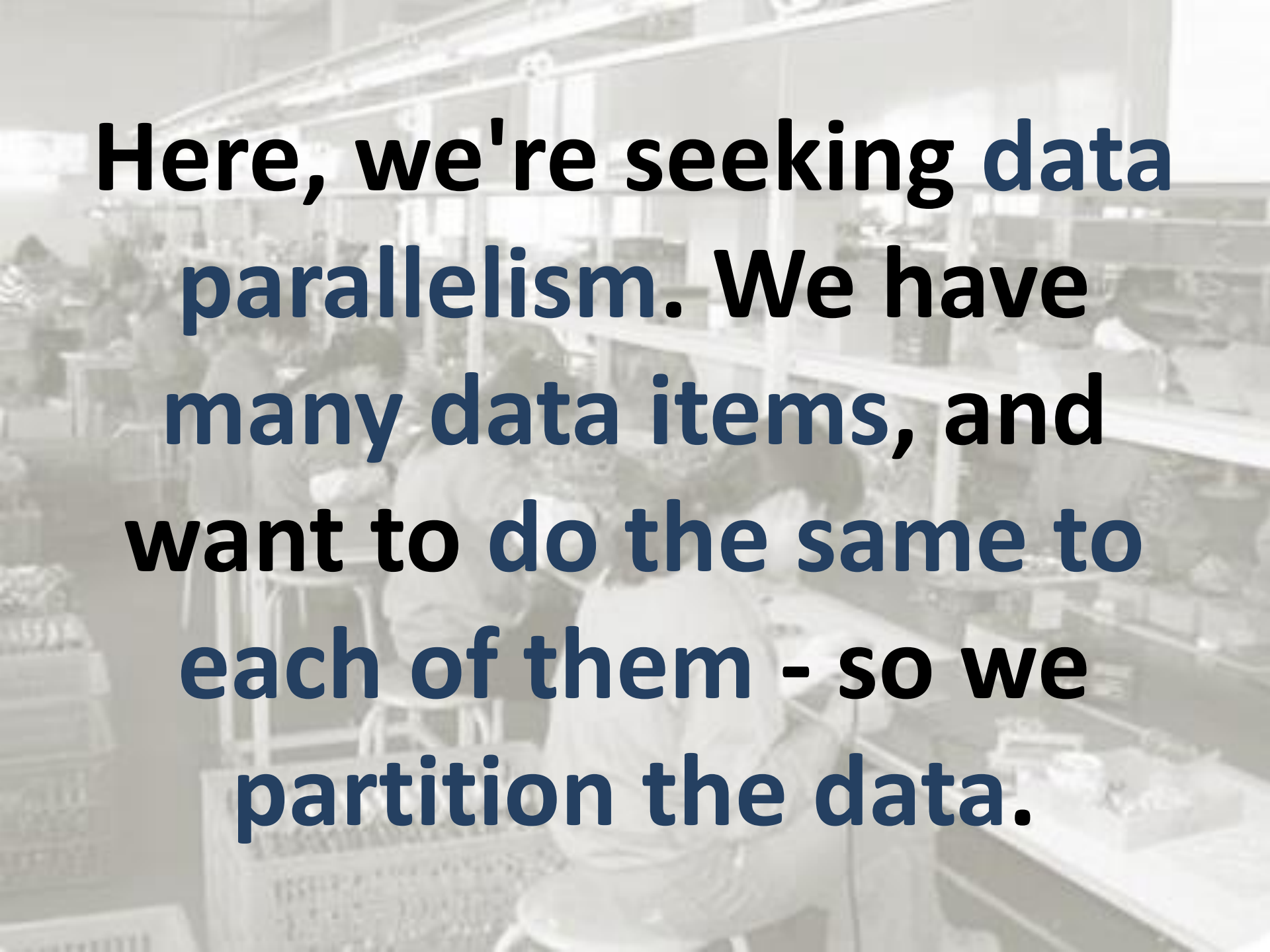
**We'll build a pipeline of operations
(hint: in Perl 6, storing the result of
things like map and grep in a Scalar
lets you talk about the pipeline
without evaluating it for results!)**

```
sub MAIN($data-dir) {  
  my $filenames = dir($data-dir);  
  my $data      = $filenames.map(&slurp);  
  my $parsed    = $data.map(&parse-climate-data);  
  my $european  = $parsed.grep(*.continent eq 'Europe');  
  my $max       = $european.max(by => *.average-temp);  
  say "$max.place() is the hottest!";  
}
```

Some utility subs omitted here...

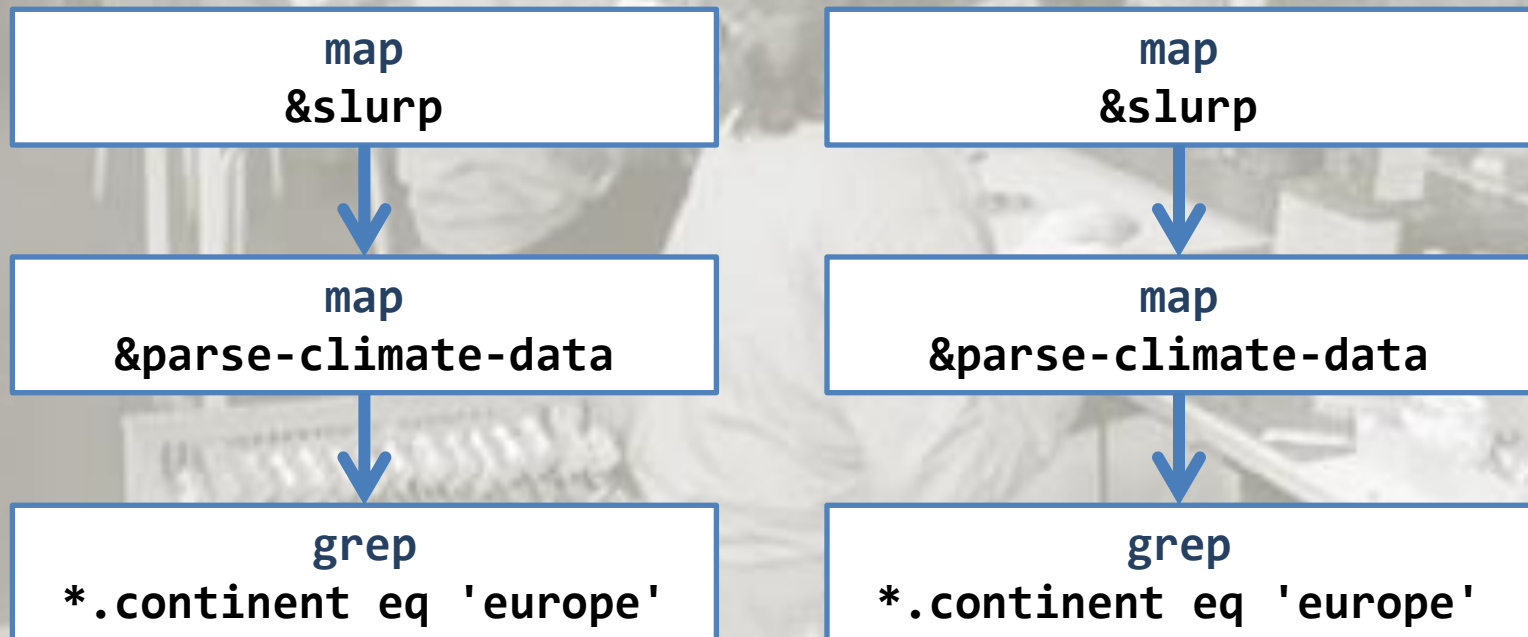
With this approach, we don't build up a lot of state in memory. One file at a time is pulled through the pipeline.





**Here, we're seeking data
parallelism. We have
many data items, and
want to do the same to
each of them - so we
partition the data.**

**We'd like to run the pipeline
on a bunch of threads,
feeding them data and
collecting the results.**





But...

**How to distribute the work and
collect the results safely?**

How to collect exceptions?

**What if keeping results ordered
relative to input matters?**

We use race to switch on parallel processing of the pipeline!

```
sub MAIN($data-dir) {  
  my $filenames = dir($data-dir).race(batch => 10);  
  my $data      = $filenames.map(&slurp);  
  my $parsed    = $data.map(&parse-climate-data);  
  my $european  = $parsed.grep(*.continent eq 'Europe');  
  my $max       = $european.max(by => *.average-temp);  
  say "$max.place() is the hottest!";  
}  
  
# Some utility subs omitted here...
```

Calling `.race()` coerces the pipeline into a parallel one. Once we reach the `max` call, multiple threads will be spawned, processing the pipeline on batches of 10 items at a time.



```
.race(batch => 32, degree => 4)
```

Run the pipeline in parallel, work in batches of 32 values at a time, and create 4 parallel workers. Produce results in whatever order they become available.

```
.hyper(batch => 64, degree => 2)
```

Run the pipeline in parallel, work in batches of 64 values at a time, and create 2 parallel workers. Make sure the results produced are relative to the order of the inputs.



.race()

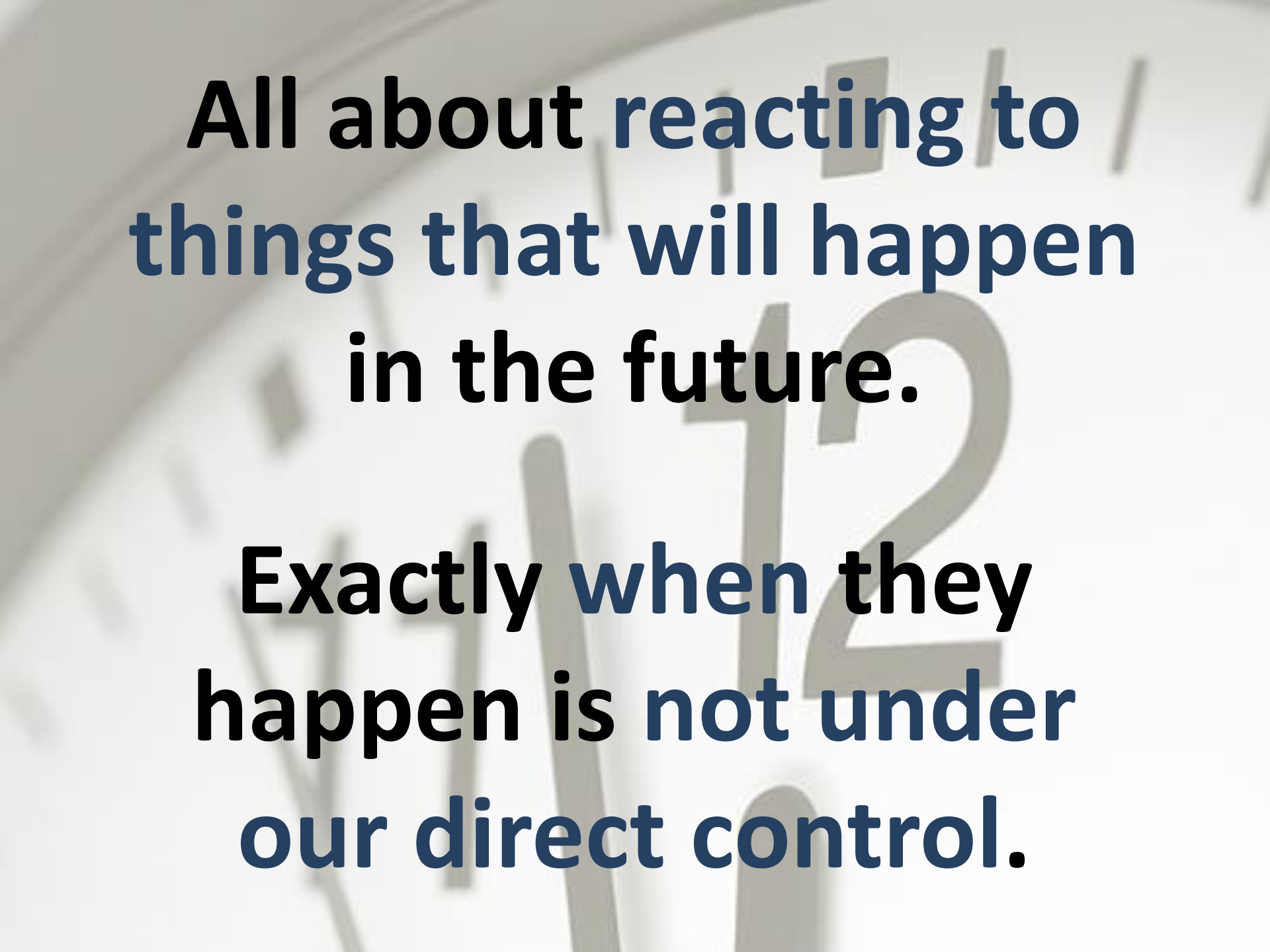
**Run the pipeline in parallel, work out the best batch size and number of workers for me.
Produce results in whatever order they become available.**

.hyper()

**Run the pipeline in parallel, work out the best batch size and number of workers for me.
Make sure the results produced are relative to the order of the inputs.**



Next up:
asynchrony



**All about reacting to
things that will happen
in the future.**

**Exactly when they
happen is not under
our direct control.**

Examples

Spawned processes completing

Responses to web requests arriving

Incoming connections to a server

User interaction with a GUI

Signals

**In some cases, we can start
an operation that will
complete in the future, and
block until it does.**

**But sometimes this doesn't
meet our needs - or won't
scale far enough.**

Example: scp all the things

We have a bunch of files we need to securely copy to many servers

We can run the scp program in a loop to upload them one at a time:

```
for @uploads -> $file-info {  
    run('scp', $file-info.local, $file-info.target);  
}
```

But how might we do 4 at a time?

Using Proc::Async

First, let's adapt our code to use Proc::Async instead (built in to Perl 6):

```
for @uploads -> $file-info {  
    my $proc = Proc::Async.new('scp',  
        $file-info.local, $file-info.target);  
    await $proc.start;  
}
```

It's **await** again! We get a **Promise** back from **\$proc.start**.

Do them all at once!

To get a step closer, we can now push each Promise onto an array, and then **await** all of them:

```
my @working;  
for @uploads -> $file-info {  
    my $proc = Proc::Async.new('scp',  
        $file-info.local, $file-info.target);  
    push @working, $proc.start;  
}  
await @working;
```

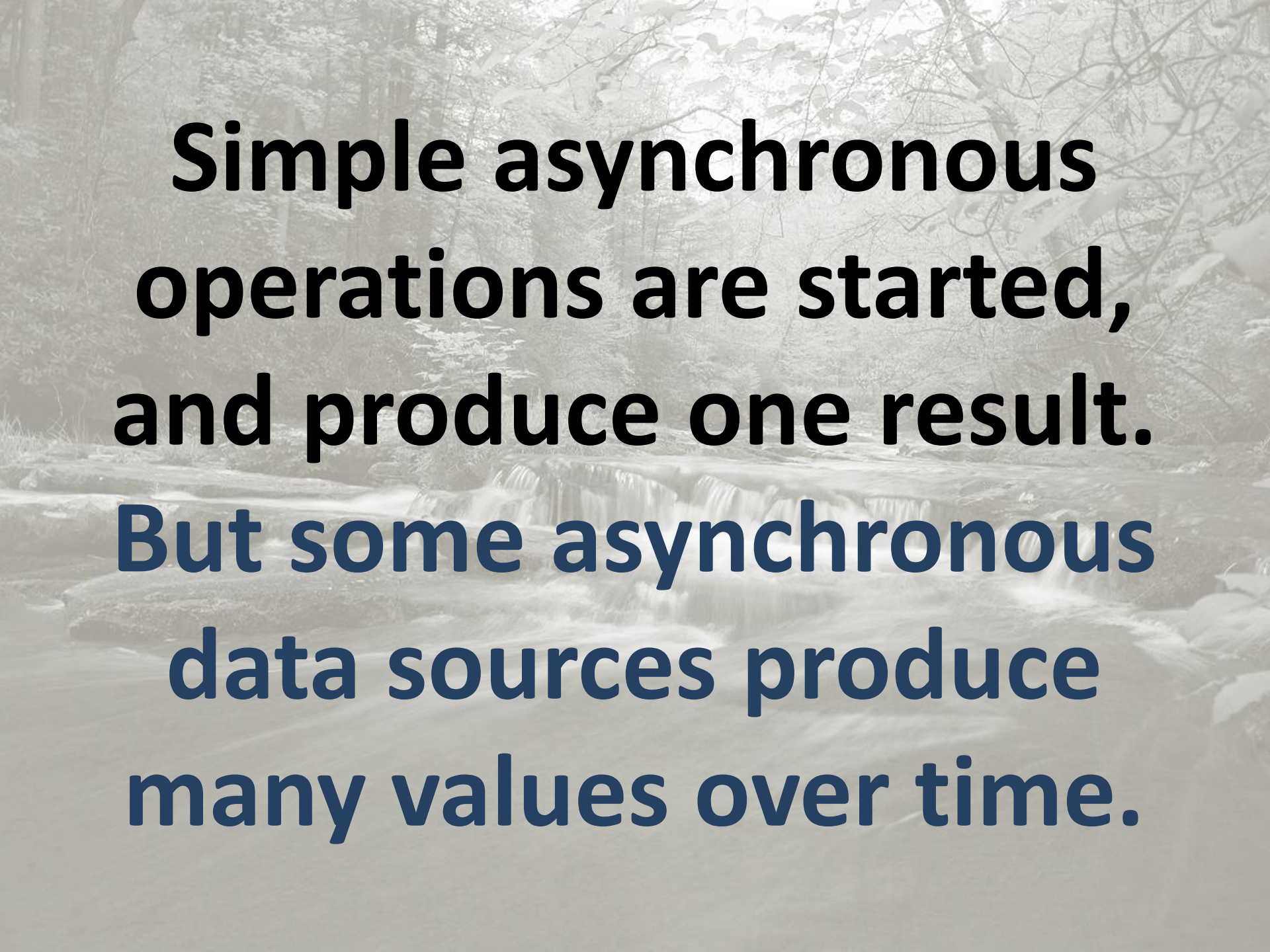
Of course, this hammers the network!

Maximum 4 at a time

If `@working` grows to 4, we wait for any Promise to be kept, and grep on unkept:

```
my @working;
for @uploads -> $file-info {
  my $proc = Proc::Async.new('scp',
    $file-info.local, $file-info.target);
  push @working, $proc.start;

  if @working == 4 {
    await Promise.anyof(@working);
    @working .= grep({ !$_ });
  }
}
await @working;
```

The background is a grayscale photograph of a forest stream with a small waterfall. The water is flowing over rocks, creating white foam. The surrounding trees and foliage are visible in the background.

**Simple asynchronous
operations are started,
and produce one result.
But some asynchronous
data sources produce
many values over time.**

Asynchronous data streams

File change notifications

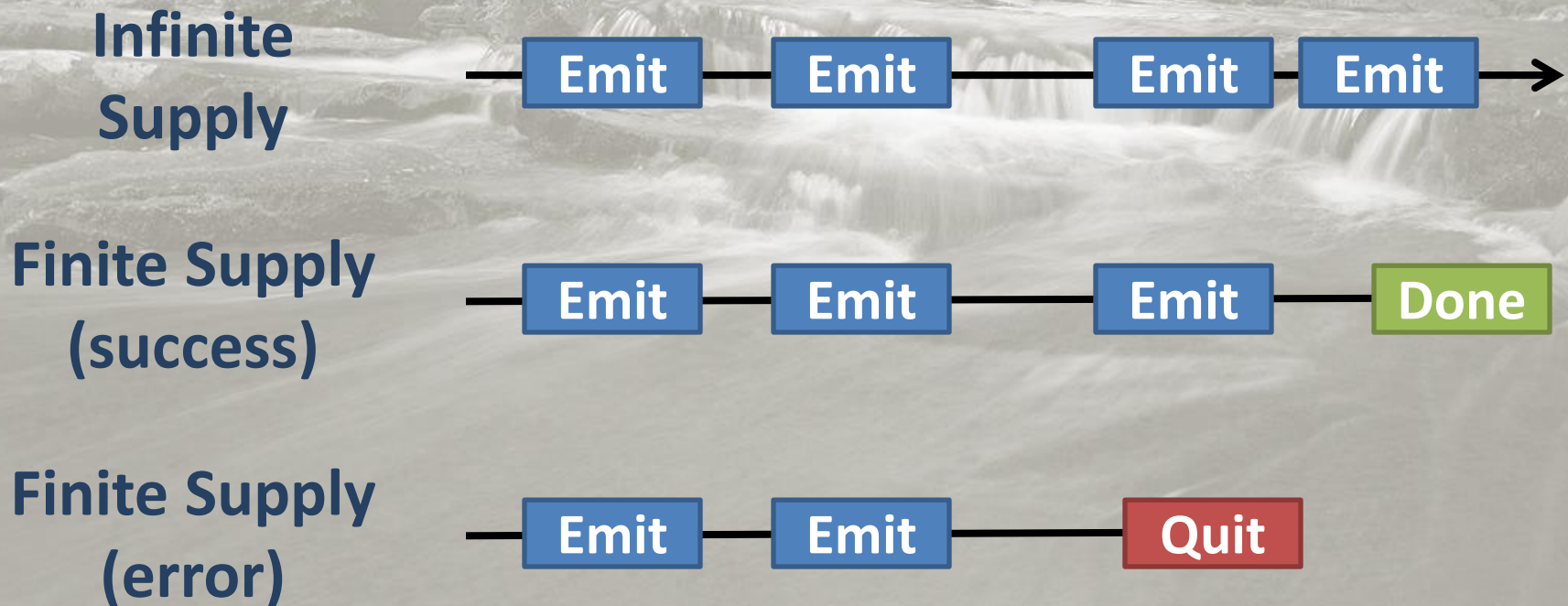
Incoming requests to a server

**Incoming packets of data to a
socket**

GUI events

Supplies

In Perl 6, an asynchronous stream of values is called a **Supply**



Automated test runner

**We'll use a file change notification
Supply to trigger automated
running of a test suite**

**We want to watch a test directory,
and optionally a number of source
directories - and should only do one
test run at a time**

File changes

The `IO::Notification` built-in provides a way to watch for changes

The `watch-path` method returns a **Supply**, which we can **tap**:

```
my $changes = IO::Notification.watch-path($test-dir);  
$changes.tap({  
  say "It changed!";  
});
```

Supply is the dual of Seq

Earlier, we set up a pipeline of operations for querying climate data. When we asked for the hottest place, max pulled values through the pipeline to work it out.

A Supply is also a pipeline, but values are instead pushed through it as they are produced at their (asynchronous) source.

Familiar methods, but async

This means we can use things like map and grep to project and filter data that arrives asynchronously. For example, we can filter by file extension:

```
my $changes = IO::Notification.watch-path($src-dir);  
my $code    = $changes.grep(*.path ~~ /<.pm .p6> $/);  
$code.tap({  
    say "A source file changed!";  
});
```

Nice, but...





Nice, but...

Most people don't solve all of their list-related problems using map, grep, and other higher order friends

Some problems are more easily expressed using for loops, if statements, etc.

But a for loop is a blocking, synchronous thing. What about asynchronous data?

whenever

Perl 6 has an asynchronous looping construct called `whenever`

The body runs whenever a value arrives:

```
whenever IO::Notification.watch-path($test-dir) {  
    maybe-run-tests('Tests changed');  
}
```

Being a loop, you can even use `LAST` to decide how to handle end of sequence!

react/supply

A whenever can live in a supply block (which can emit values) or a react block (works like entering an event loop):

```
my $code = supply {  
  whenever IO::Notification.watch-path($src-dir) {  
    emit .path if .path ~~ /<.pm .p6> $/;  
  }  
}  
react {  
  whenever $code -> $path {  
    say "Code file $path changed!";  
  }  
}
```

Back to our test runner

On changes to the specified test and source directories, maybe run the tests:

```
sub MAIN($test-dir, *@source-dirs) {  
  react {  
    whenever IO::Notification.watch-path($test-dir) {  
      maybe-run-tests('Tests changed');  
    }  
    for @source-dirs -> $dir {  
      whenever IO::Notification.watch-path($dir) {  
        maybe-run-tests('Source changed');  
      }  
    }  
    ...  
  }  
}
```

Deciding whether to run

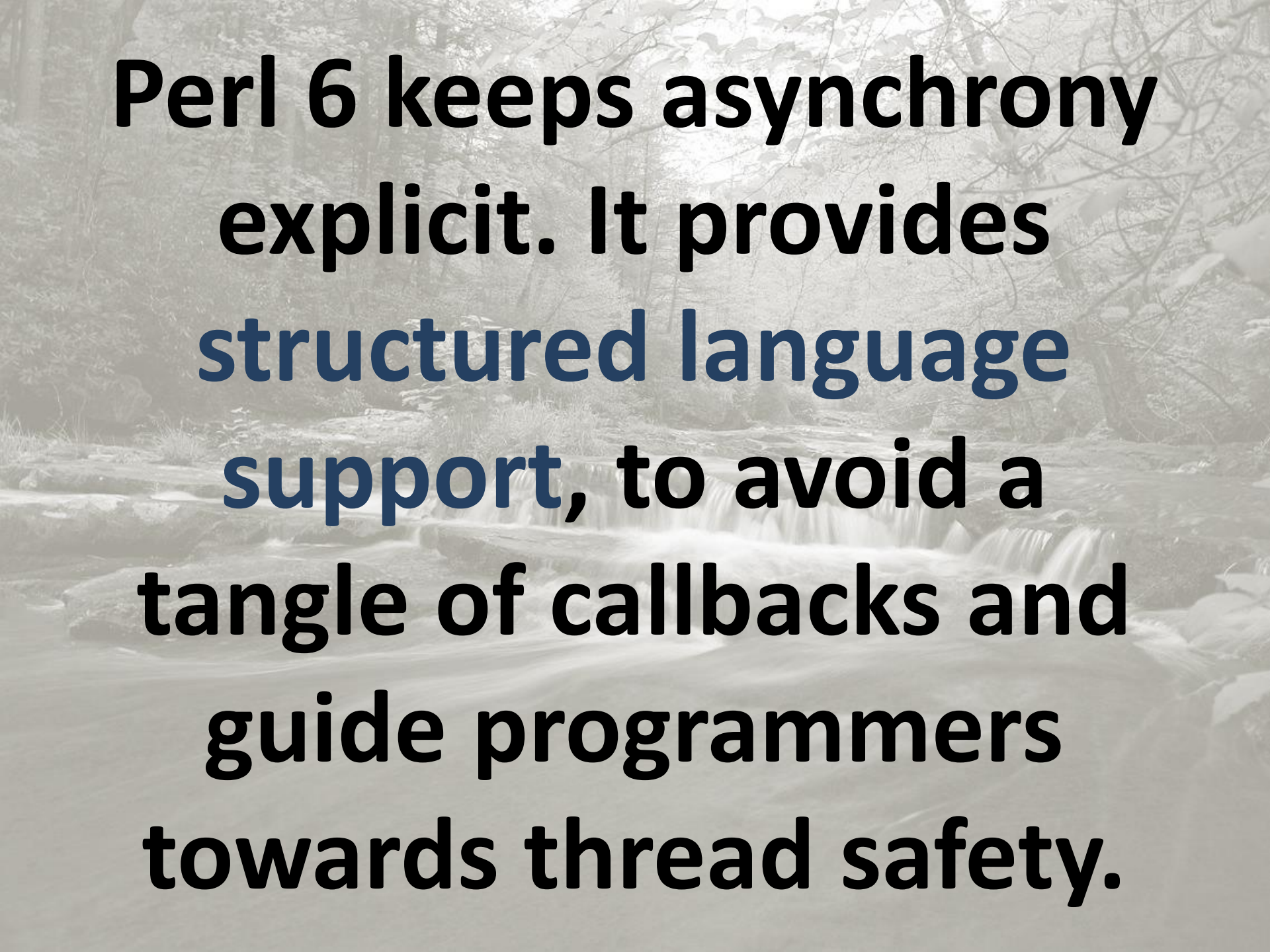
The notifications may arrive on different threads - but only one thread may be in a supply/react at once - so this is safe:

```
sub maybe-run-tests($reason) {  
  state $running-tests = False;  
  unless $running-tests {  
    say "Running tests ($reason)";  
    $running-tests = True;  
    whenever run-tests() {  
      print "\n\n";  
      $running-tests = False;  
    }  
  }  
}
```

Actually doing the running

It's our old friend, `Proc::Async` again.
We output `STDOUT` indented and discard `STDERR`. We return a `Promise`; whenever can work fine against those too.


```
sub run-tests() {  
  my $runner = Proc::Async.new('prove ...');  
  whenever $runner.stdout -> $output {  
    print $output.indent(2);  
  }  
  whenever $runner.stderr { } # Discard  
  return $runner.start;  
}
```

A background image of a forest stream with a small waterfall, overlaid with text. The text is in a bold, sans-serif font, with the words 'structured language support' in a darker blue color and the rest in black. The text is centered and occupies most of the frame.

**Perl 6 keeps asynchrony
explicit. It provides
structured language
support, to avoid a
tangle of callbacks and
guide programmers
towards thread safety.**



**Finally:
concurrency**



**Concurrency is about
competition to access
and mutate some
shared resource.**



**Passengers checking in for a flight
at the same time must not be
able to choose the same seat!**

A simple seat allocator

```
class Flight {  
  has %!seats;  
  
  submethod BUILD(:@seat-labels) {  
    %!seats{@seat-labels} = False xx *;  
  }  
  
  method choose-seat($seat, $passenger-name) {  
    die "No such seat" unless %!seats{$seat}:exists;  
    die "Seat taken!" if %!seats{$seat};  
    %!seats{$seat} = $passenger-name;  
  }  
}
```

But...

The code contains a **data race**!

```
method choose-seat($seat, $passenger-name) {  
  die "No such seat" unless %!seats{$seat}:exists;  
  die "Seat taken!" if %!seats{$seat};  
  %!seats{$seat} = $passenger-name;  
}
```

If two threads are in this method at the same time with the same `$seat`, they may both see the seat is not taken, and then place their passenger into it!

Solution: a monitor

A monitor is a class where only one thread may be running a method on a particular instance at a time

So, the second passenger wanting to select a seat would have to wait until the first has finished selecting theirs

How do we refactor our code?

It's easy!

```
use OO::Monitors;

monitor Flight {
  has %!seats;

  submethod BUILD(:@seat-labels) {
    %!seats{@seat-labels} = False xx *;
  }

  method choose-seat($seat, $passenger-name) {
    die "No such seat" unless %!seats{$seat}:exists;
    die "Seat taken!" if %!seats{$seat};
    %!seats{$seat} = $passenger-name;
  }
}
```

It's easy!

```
use OO::Monitors;
```

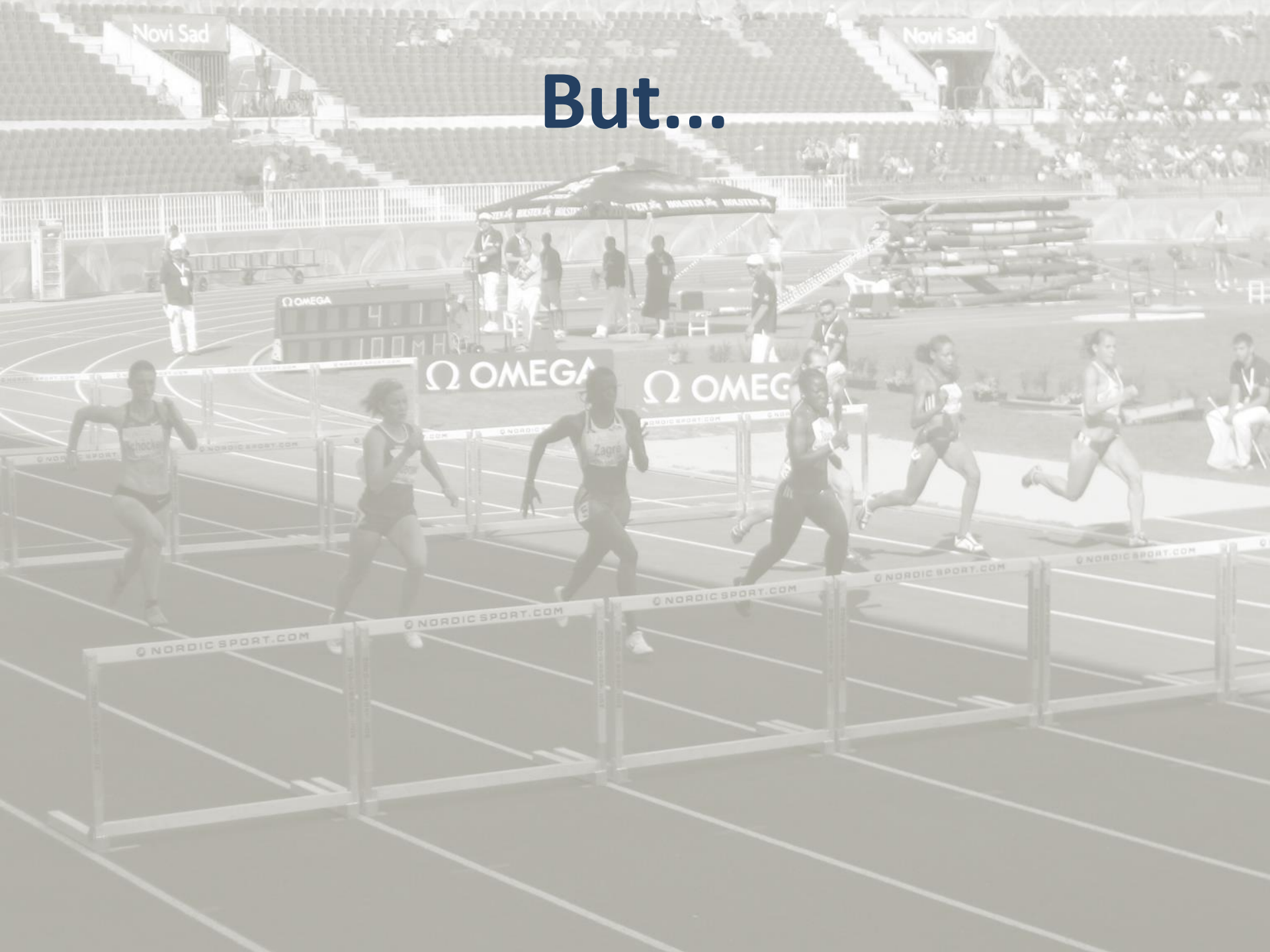
```
monitor Flight {  
  has %!seats;
```

New kind of package, provided
by OO::Monitors module!

```
  submethod BUILD(:@seat-labels) {  
    %!seats{@seat-labels} = False xx *;  
  }
```

```
  method choose-seat($seat, $passenger-name) {  
    die "No such seat" unless %!seats{$seat}:exists;  
    die "Seat taken!" if %!seats{$seat};  
    %!seats{$seat} = $passenger-name;  
  }  
}
```

But...



The background is a grayscale photograph of a track and field stadium. In the foreground, several hurdles are visible on a running track. Athletes are seen in motion, some jumping over the hurdles. In the background, there are tiered seating areas (stands) and banners for 'Novi Sad' and 'OMEGA'.

But...

Suppose we use a monitor in an asynchronous web application

In the case of contention, one of the processing threads will block synchronously waiting for the other thread to leave the monitor

Can we do better?

Use an actor instead!

```
use OO::Actors;

actor Flight {
  has %!seats;

  submethod BUILD(:@seat-labels) {
    %!seats{@seat-labels} = False xx *;
  }

  method choose-seat($seat, $passenger-name) {
    die "No such seat" unless %!seats{$seat}:exists;
    die "Seat taken!" if %!seats{$seat};
    %!seats{$seat} = $passenger-name;
  }
}
```

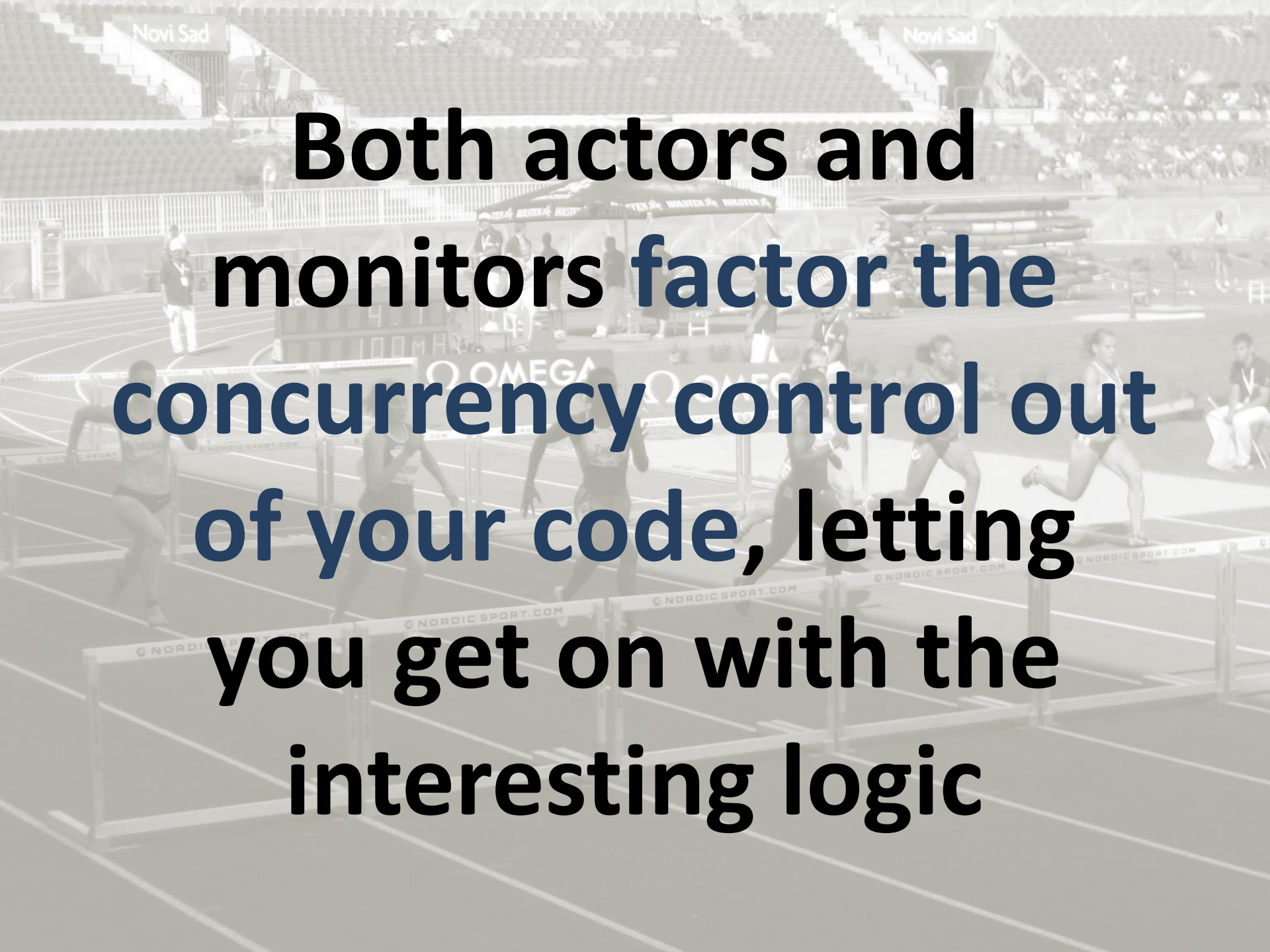
What an actor does

An actor puts incoming method calls into a processing queue

Method calls on an actor return a Promise, which the caller awaits:

```
await $flight.choose-seat($seat, $passenger-name);
```

The processing thread is free to deal with other requests in the meantime!

The background is a grayscale photograph of a hurdle race in progress at a large stadium. Several athletes are captured mid-stride, jumping over hurdles. The hurdles have 'NORDIC SPORT.COM' printed on them. In the background, there are banners for 'OMEGA' and 'WALTER'. The stadium seating is visible, with 'Novi Sad' written on some of the upper tiers.

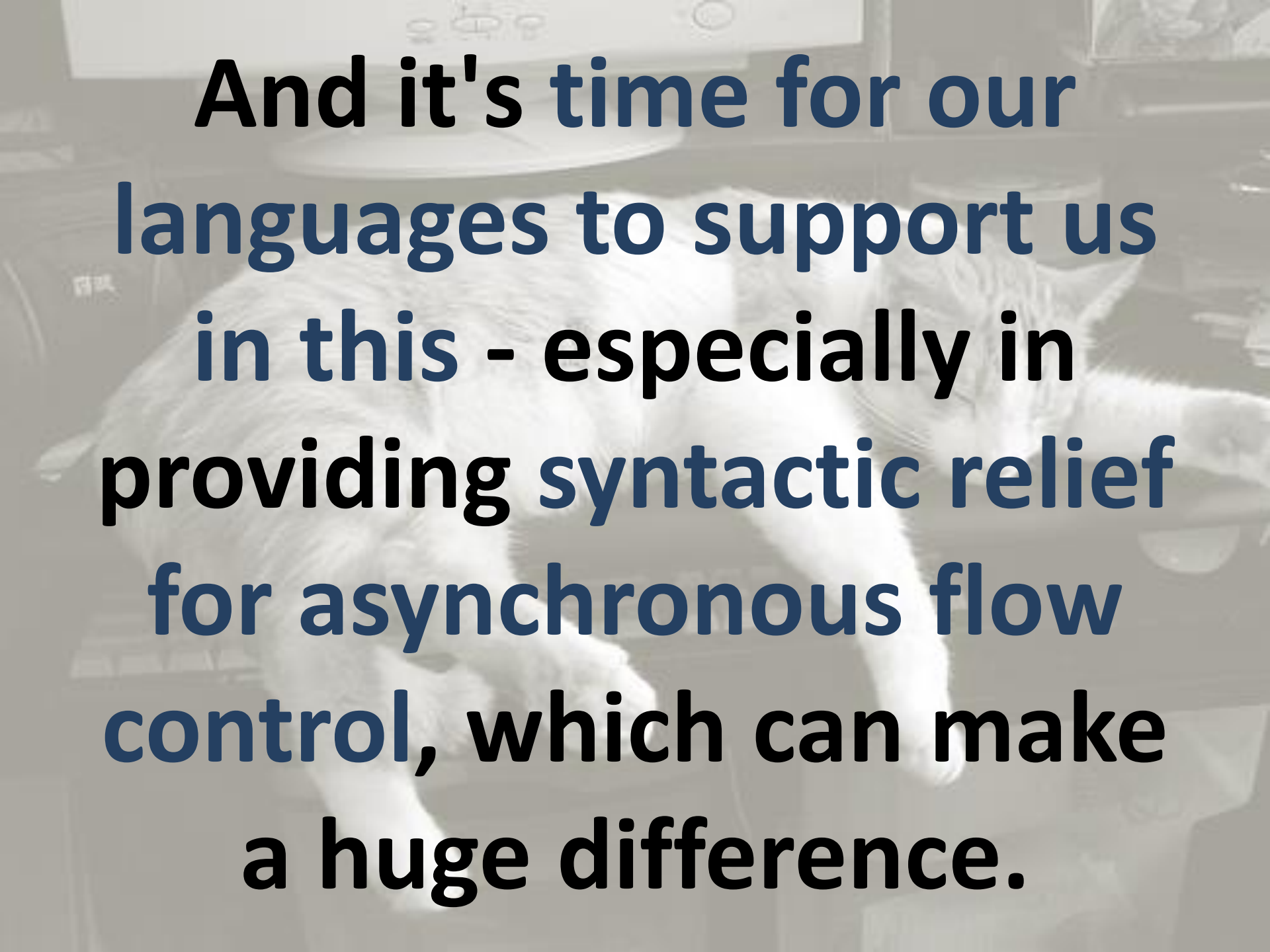
**Both actors and
monitors factor the
concurrency control out
of your code, letting
you get on with the
interesting logic**

A photograph of a light-colored cat with dark stripes, lying down on a dark desk. The cat is positioned horizontally, with its head to the right and its front paws extended forward. In the background, a computer monitor is visible on the left, and a keyboard is partially visible below the cat. The entire image has a semi-transparent dark overlay, and the text "In closing..." is centered over the cat's body.

In closing...

A white cat is lying on a desk, looking towards the camera. In the background, a computer monitor is visible. The text is overlaid on the image.

**It's time to embrace
structured approaches to
parallel, asynchronous,
and concurrent
programming.**

A white cat is lying on a desk, looking towards the camera. In the background, a computer keyboard and mouse are visible. The text is overlaid on the image in a large, bold, blue font.

**And it's time for our
languages to support us
in this - especially in
providing syntactic relief
for asynchronous flow
control, which can make
a huge difference.**

A white cat with grey patches is curled up and sleeping on a dark desk. In the background, a computer monitor is visible. The text "Thank you!" is overlaid in a dark blue, bold, sans-serif font.

Thank you!

Any questions?