Concurrency, Parallelism and Asynchrony: Perl 6 plotting and prototyping

Jonathan Worthington
Why should we care?

In the last decade, hardware has become increasingly parallel, as getting more cycles out of a single core has started to run into physical limitations.

→ Growing need to do parallel programming

At the same time, distributed systems have become the norm, and users have grown to expect ever-more responsive applications; see the recently published Reactive Manifesto for discussion on this.

→ Growing need for asynchrony
Why now?

Perl 6 has, from the start, aimed to provide better support for parallel programming, a Perl 5 weak area; in reality, little visible progress up to now

By contrast, Perl 5 already has good answers on asynchrony; again, little to show from Perl 6 land

Within the last few months, Rakudo - the most complete Perl 6 implementation - has been able to run on the JVM

We now have all the parallel, concurrent and asynchronous primitives we need to move forward
Why me?

There's something that not all in the Perl world know
Why me?

There's something that not all in the Perl world know

I'm also a C# developer/teacher at my dayjob
There's something that not all in the Perl world know

I'm also a C# developer/teacher at my $dayjob

Coping with threads is a daily reality for many C# developers, though not all cope especially well 😊

Parallel and concurrent programming has shown up in many projects and at many clients ➔ I get to observe lots of interesting ways to screw it up!

Also been teaching the new asynchronous C# features
Parallelism

Break a problem into pieces we can do at the same time

This enables us to exploit multi-core CPUs

We can know we're free of parallelization bugs if the result is always the same as if we'd computed it serially
Asynchrony

Synchronous is "the normal thing": we call something, it does its work, then returns a result

With asynchronous: we call something, it sets the work in motion, and returns (typically, an object that represents the ongoing work)
Concurrency

About coping with events arising whenever then please, and trying to do "the right thing"

Happens inside of parallelism (different pieces of the work will complete at different times), but that's the easy case, as there's a correctness criteria

In an inherently concurrent domain with many autonomous actors, correctness less obvious
TMTOWTDI - but they compose

In Perl, we're used to mixing different paradigms to solve problems, combining them effortlessly

```perl
my @hammers = @tools.grep({ .name ~~ /hammer/ });
```
In Perl, we're used to mixing different paradigms to solve problems, combining them effortlessly.

```perl
my @hammers = @tools.grep({ .name ~~ /hammer/ });
```
In Perl, we're used to mixing different paradigms to solve problems, combining them effortlessly.

```
my @hammers = @tools.grep({ .name ~~ /hammer/ });
```

Likewise, we may need to employ parallelism, asynchrony and concurrency in our system.

They solve different problems, but should compose.
Threads and locks

These are the assembly language of parallel and concurrent programming

ült Used directly, they rarely compose well 😞

A component spawning two worker threads may be OK in isolation, but what if your application needs to use 20 components that do this?

Two pieces of code that use locks can each work reliably in isolation, but may have a deadlock risk if used together
Threads: the Perl 6 take

"Make the easy things easy and the hard things possible"

Threads and locks are a hard thing

We won't stop you writing...

my $t1 = Thread.start({ do_hard_computation() });
# ...
$t1.join();

...and getting yourself an OS-level thread

But it's a last resort, not a first resort
A more Promise-ing approach

An async code block schedules a piece of work to be done asynchronously (on some other thread)

```perl
my $p10000 = async {
    (1..Inf).grep(*.is-prime)[9999]
}
```

It produces a Promise object, which represents the ongoing piece of work

We don't spawn a thread per async block! The spawning of threads is managed by a scheduler, meaning it can spread work over a sensible number of them.
Promise basics

The current status of a Promise (Planned, Running, Kept, Broken) can be obtained with the `status` method.

```plaintext
say $p10000.status;
```

The `result` method obtains the result produced by the async block, unless it died, in which case the exception will be rethrown so it can be handled.

```plaintext
say $p10000.result;
```

Calling `result` on a Promise that is not yet Kept or Broken will block until the Promise's execution is done.
The await function takes one or more Promise objects, blocks until they all have results, and returns a list of their results, throwing any exceptions.

Naively, calls result, but allowed to be smarter (yield)
A more useful example

Here's a program that adds up the number of lines across all the files in a given directory

```plaintext
say [+] dir('docs/announce').map({
    .IO.lines.elems
});
```

Could instead have a Promise to process each file, then await them and add up the results

```plaintext
say [+] await dir('docs/announce').map({
    async {
        .IO.lines.elems
    }
});
```
Promise combinators

One of the most powerful things we can do with Promises is write **combinators** that act on one or more of them, producing some kind of aggregate Promise.

The **anyof** and **allof** combinators produce a Promise that is kept when one or all of a set of promises are kept.

```perl
my @promises = dir('docs/announce').map({
    async {
        .IO.lines.elems
    }
});

my $lines_counted = Promise.allof(@promises);
```
The Promise class also has a `sleep` method, which produces a promise that will be kept after a delay.

Unlike the normal `sleep` function, `Promise.sleep` will not block a thread. You can have hundreds of them.

Used with `anyof`, we have a timeout mechanism:

```javascript
await Promise.anyof($p2000, Promise.sleep(5));
say $p2000.status == Kept ?? $p2000.result !! 'Timed out';
```
After the Promise, then...

The **then** method on a Promise registers a piece of code to run when the promise is kept or broken.

Most significantly, **this method also returns a Promise** that represents the composite piece of work.

Thus, we can implement sleep sort as...

```perl
my @a = (1..20).pick(*);
await @a.map(-> $n { Promise.sleep($n).then({ say $n }) })
```
Make your own Promise

While an async block produces a Promise backed by code scheduled on the thread pool, you can put *anything* that will later produce a value or exception behind one

Simply create a new Promise...

```
my $p = Promise.new;
```

...and then call either *keep* or *break* some point later:

```
$p.keep($value);
```

Has a *then*, can participate in combinators, etc...
Example: nth_or_timeout (1)

Making our own Promise objects that we keep or break is useful for implementing new combinators.

Our timeout mechanism earlier sucked because the computation continued even after the timeout, and we had to introspect the timeout Promise 😞

Would be nice to just have written:

```plaintext
say await nth_or_timeout((1..Inf).grep(*.is-prime), 2000, 10);
```
sub nth_or_timeout(@source, $n, $timeout) {
    my $p = Promise.new;
    my $t = Promise.sleep($timeout);
    ...
    $p
}
Example: nth_or_timeout (3)

```perl
sub nth_or_timeout(@source, $n, $timeout) {
    my $p = Promise.new;
    my $t = Promise.sleep($timeout);
    async {
        my $result;
        ...
        $p.keep($result);
    }
    $p
}
```
sub nth_or_timeout(@source, $n, $timeout) {  
  my $p = Promise.new;  
  my $t = Promise.sleep($timeout);  
  async {  
    my $result;  
    for ^$n {  
      if $t.status == Kept {  
        $p.break('Timed out');  
        last;  
      }  
      $result = @source[$n];  
    }  
    $p.keep($result);  
  }  
  $p  
}
Beyond scalars

Promises are all about a single value (or the failure to produce one due to an error)

However, in Perl we don’t just have scalars, but also arrays and hashes

A `Channel` represents a set of values delivered asynchronously

A `KeyReducer` represents a hash whose keys and values are contributed asynchronously
Channels

Provide a thread-safe synchronization mechanism based around a queue

A channel is created like this:

```
my $c = Channel.new;
```

Work happening on one or more threads can send:

```
$c.send($result);
```

Meanwhile, one or more others can receive:

```
my $val = $c.receive;
```
A recent example from my work involved a conveyor belt of agricultural product (maybe wheat) having moisture content readings arriving from a sensor at irregular intervals (often a few a second, sometimes a whole second between them). We may simulate it as:

```haskell
my $belt_chan = Channel.new;
async {
    loop {
        $belt_chan.send(rand xx 100);
        await Promise.sleep((0.15, 0.25, 1).pick);
    }
}
```
Channels example (2)

Something else can receive these readings and do the required calculation on them

Here, we just do an average, but of course the actual work being done in the real world system is vastly more complicated!

```
loop {
    my @values = $belt_chan.receive;
    say [+](@values) / @values.elems;
}
```

Note that receive is a blocking operation
The select function (1)

Often, there are multiple channels that may be producing interesting values, and you want to take action based on whichever one has a value available.

For example, it may be that occasionally a control reading is taken by analyzing a scoop of the product:

```perl
my $sample_chan = Channel.new;
async {
  loop {
    await Promise.sleep((5, 10, 15).pick);
    $sample_chan.send(rand);
  }
}
```
The select function takes a list of pairs, mapping channels to a closure to execute if the channel can receive

```plaintext
loop {
    select(
        $belt_chan => -> @values {
            say "BELT: {+[(@values) / @values.elems]}";
        },
        $sample_chan => -> $sample {
            say "SAMPLE: $sample";
        }
    );
}
```
select works on Promises too

You can also use select to take different action depending on which Promise gets kept/broken first, or even do a mixture of channels and promises

```perl
my $run_ends = Promise.sleep(30);
loop {
    select {
        $belt_chan => -> @values { ... },
        $sample_chan => -> $sample { ... },
        $run_ends => -> $ {
            say "End of measurement!";
            exit;
        }
    );
}
```
Channel combinators (1)

Just like we could write Promise combinators, such as `nth_or_timeout`, we can also write channel ones.

For example, we may like to write one that collects the belt measurements made in 2 seconds and send them as a group, on some other channel.

```perl
my $b2s_chan = collect_per_interval($belt_chan, 2);
loop {
    sub avg(@m) { [+](@m) / (@m.elems || 1) } 
    my @measurements = $b2s_chan.receive;
    say avg(@measurements.map(&avg)) ~
        " (from @measurements.elems() measurements)";
}
```
sub collect_per_interval(Channel $chan, $seconds) {
  my $res = Channel.new;
  ...
  $res
}
sub collect_per_interval(Channel $chan, $seconds) {
    my $res = Channel.new;
    async {
        my $period = Promise.sleep($seconds);
        my $accum = [];
        ...
    }
    $res
}
sub collect_per_interval(Channel $chan, $seconds) {
    my $res = Channel.new;
    async {
        my $period = Promise.sleep($seconds);
        my $accum = [];
        loop {
            select(
                $chan => -> $msg { $accum.push($msg) },
                $period => -> $ {
                    $res.send($accum);
                    $accum = [];
                    $period = Promise.sleep($seconds);
                }
            )
        }
    }
    $res
}
The KeyReducer

Promise is a synchronization mechanism for single values, and Channels are a synchronization mechanism for sequences of values, either finite or infinite.

What if you want to have many parallel workers producing associative (key/value) data, and then collapse it down to a single hash?

This is what the KeyReducer is for. It's a way to have work going on in many threads contribute to a hash, which can be snapshotted or reach some final state at which point no more contributions are allowed.
We need to count the number of occurrences of different variable names across many source files.

Want to do the counts per file in parallel, and then safely incorporate them into a single result hash.

First value just goes into the result hash, followup ones just get summed.

```perl
my $var_counts = KeyReducer.new(
    -&gt; $first, { $first },
    -&gt; $cur, $next { $cur + $next });
```
Process each file in an async block, counting its variables then contributing them to the overall result

```perl
await dir('src/core').map(-> $file {
    next if $file.d;
    async {
        my %results;
        my $src = slurp($file);
        for $src.match(/<[$%@]> \w+/, :g) -> $var {
            %results{~$var}++;
        }
        $var_counts.contribute(%results);
    }
});
```
Finally, obtain the result hash, sort its pairs by the number of variable occurrences descending, and output the name and count

```
for $var_counts.result.sort(-*.value) -> $res {
  say "$res.key() is used $res.value() times"
}
```

Our overall approach is really just a form of map-reduce, with the mapping distributed over multiple CPU cores.

Promise, Channel and KeyReducer are synchronization primitives for different shapes of data.
Asynchronous I/O

Those of you familiar with asynchronous I/O will have realized that many of these examples are going to end up clogging up the thread pool by doing blocking I/O.

We'd prefer to issue the requests to read files into memory asynchronously, and only execute work in the thread pool once we have the data.

Rakudo on JVM is in the process of gaining support for asynchronous I/O, and there is enough in place to revisit our previous example and improve it 😊.
Basic asynchronous I/O example

The slurp method on IO::Async::File returns a Promise that, if/when kept, has the file contents.

Can use the familiar then method on it:

```perl
await dir('src/core').map({
    next if $path.d;
    IO::Async::File.new($path).slurp().then({
        my %results;
        for $f.result.match(/<[$%@]>\w+/, :g) -> $var {
            %results{~$var}++;
        }
        $var_counts.contribute(%results);
    });
});
```
Asynchronous I/O: much to come

What's there so far is very much a work in progress

So far you can only slurp (which gives a Promise) or lines (which gives a Channel to which each line is sent)

Really, though, it's asynchronous I/O on sockets that is the really interesting thing

Getting sockets ported has been one of the things left fairly late in the Rakudo on JVM work (something has to be), but it's underway now; should have something to see on this within the next couple of months
Other things in the pipeline

Hyper-operators for data-parallel operations

The hyper and race list contexts, which process map, grep and so forth in parallel (also data-parallel)

Feed operators for setting up producer/consumer chains (probably just a convenient sugar for a Channel use case)

Exposing various lower-level primitives – not for direct use by the everyday programmer, but for those building the higher level pieces or with special requirements
Closing thoughts

There's no "one true way" in this area, which fits well with the Perl mindset – but being able to compose our use of different parallel and async operations matters.

Shared memory is sometimes useful, but experience shows that it's all too often screwed up.

By contrast, higher level synchronization tools (promises, channels, reducers) come with simpler usage rules and typically lead to more composable solutions.

This is only the beginning; stay tuned!
Thank you!

Questions?

Blog: 6guts.wordpress.com
Twitter: @jnthnwrthngtn
Email: jnthn@jnthn.net