HTEX interchange in 3 languages

Ben Clifford
fun project 2025

structure:
21 slides

so 3 languages and 1 overview

5 slides per language
and
6 slides for overview

OR

6 slides per language
and
3 slides for overview (introduce more stuff as-we-go-along)

A language that doesn't affect the way you think about programming, is not worth knowing

-- Alan Perlis, SIGPLAN Notices, 1982

US-RSE slack channels:

R
Julia
Rust
Python
JavaScript
Mojo

This talk:
Rust
Elixir
Idris2

My professional life:
Java
C
PHP
Fortran
Groovy
Dart
LambdaMOO
Python
Visual Basic

I'm going to talk about work I've done with different languages and some of the fun I've had with Parsl and some influences on production Parsl

This is a talk of me screwing around, not me being paid to do something.

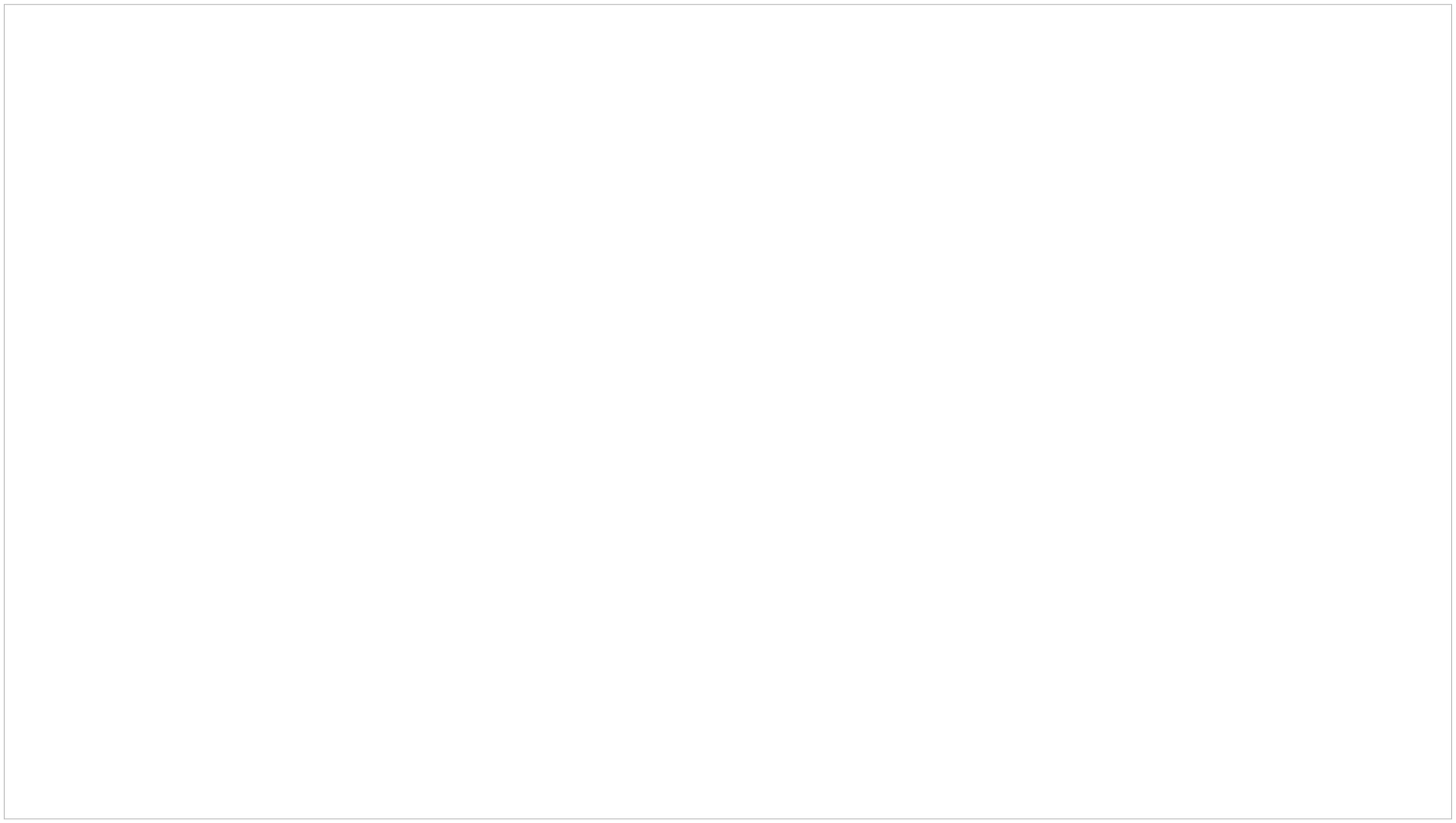
I'm interested in various programming languages "for the sake of it"
over my paid career I've worked in... Java, C, PHP, Haskell, Fortran, groovy, dart, LambdaMOO, Python (in arbitrary order), Visual BASIC
interested in other/all languages

why care about different languages? they're all turing-complete / turing-equivalent: if you can write a program in one, you can write it in any other - in that turing completeness sense

more interesting is human expressivity: can i write things easily / readably / maintainably?

US RSE slack channels: #R, #rust, #julia, #python
(as an example of what the parsl-adjacent community might be expected to use) #javascript #mojo

lots of stuff in Parsl/GC stack is not in Python, despite GC/Parsl being Python-oriented products:
libzmq is C
rabbitmq is Erlang
? numerics in fortran in many apps?



talk about parsl inter-process *protocols* as a thing to design - rather than just happening implicitly.
in the context of discussing different pieces of parsl running on different versions (of Parsl and of Python)

motivated by a throwaway comment I made about thinking about this: regard the component at the other end of a connection or message path as written in an arbitrary other language - which happens to be quite like the impl we are working on, but not quite the same - because its a different Python or a different Parsl or both. So dont' let that similarity fool you into complacency. not because I think we should be rewriting pieces of production Parsl in other languages.
so "you should be able to rewrite the htex interchange in rust"

... but then? what would this really look like?
to rewrite the interchange in rust?

[meta]

generated a bunch of parsl issues and PRs (by me and by others)

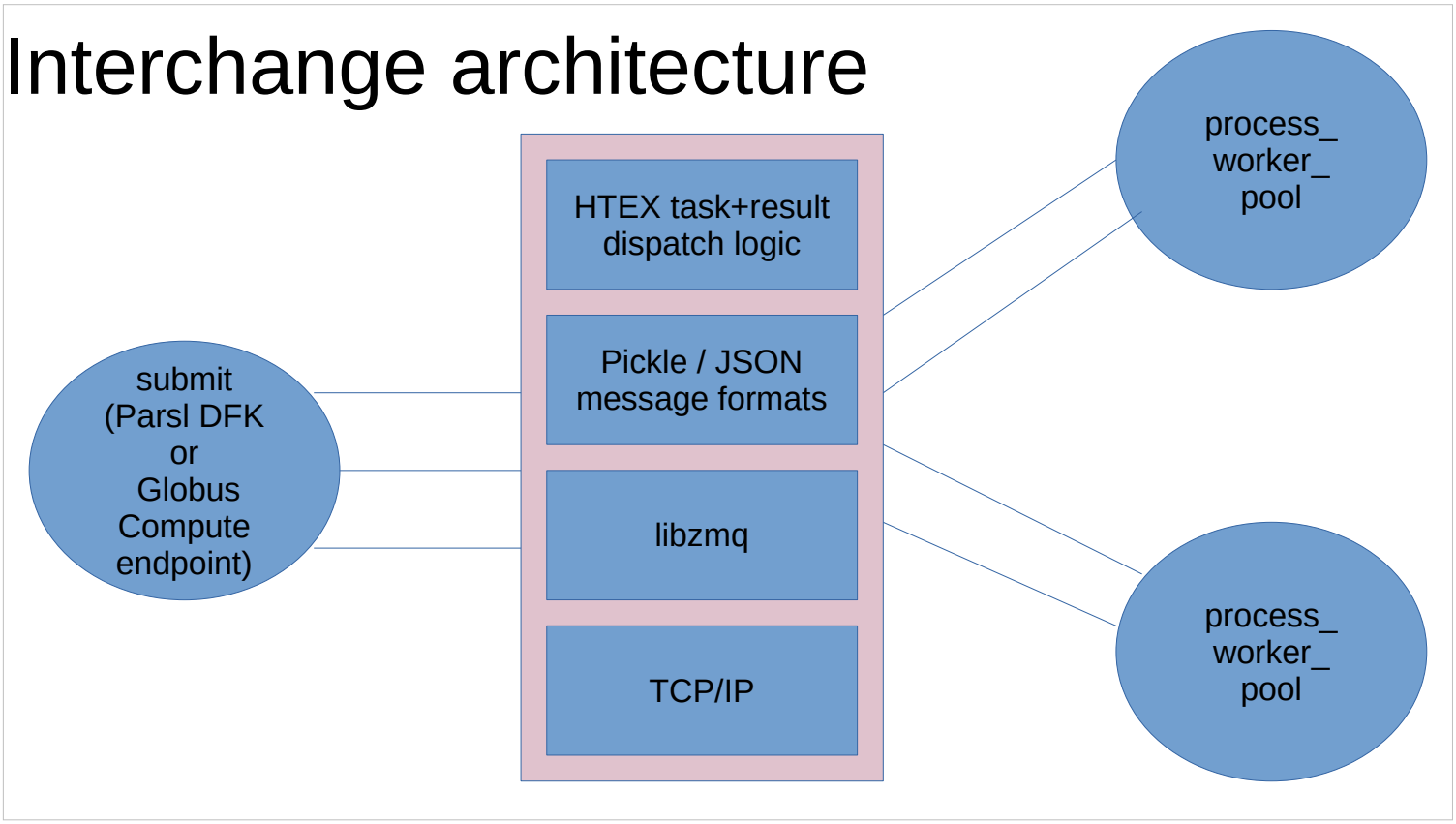
some real changes in production parsl motivated by this

list them here but get them spread through the talk in context, ideally?

kevin’s PR #3871 / my issue #3022 - one zmq channel between workers and interchange. this is after most of my work, so this is the end of me keeping this code up to date with parsl master June 2025

framing jumble: pickle lists, multipart zmq messages, multiple single part zmq messages - is there an issue(s) for that? how to fit into language context? just that serde made me pay attention to this more, rather than it being mostly abandoned code from years ago - so then on the serde slide for rust language?

manager ID is .. .a byte string .. .that is actually a fixed length ASCII encoding of a hex piece. that is then used for OS filenames etc. so what is actually valid here? Should we be fuzz testing? what happens if i put ../whatever/.... ?



interchange architecture as far as this talk is concerned

[interchange is a separate unix process. i hacked the code to use arbitrary command line, then Reid did it properly in PR #NNNN]

task + worker pool / dispatch - interchange logic
Pickle/JSON message format - built into Python
ZMQ - libZMQ
TCP/IP - OS

[JSON was removed in PR #NNNN by Kevin]
pickle is a whole adventure in itself - see PyCon LT 2024 https://www.youtube.com/watch?v=Qn-1hGLrzR0 The Ghosts of Distant Objects

submit side (dataflow kernel or endpoint)

worker side
[worker side - 2 connections merged into 1 connection by Kevin in PR #NNNN]

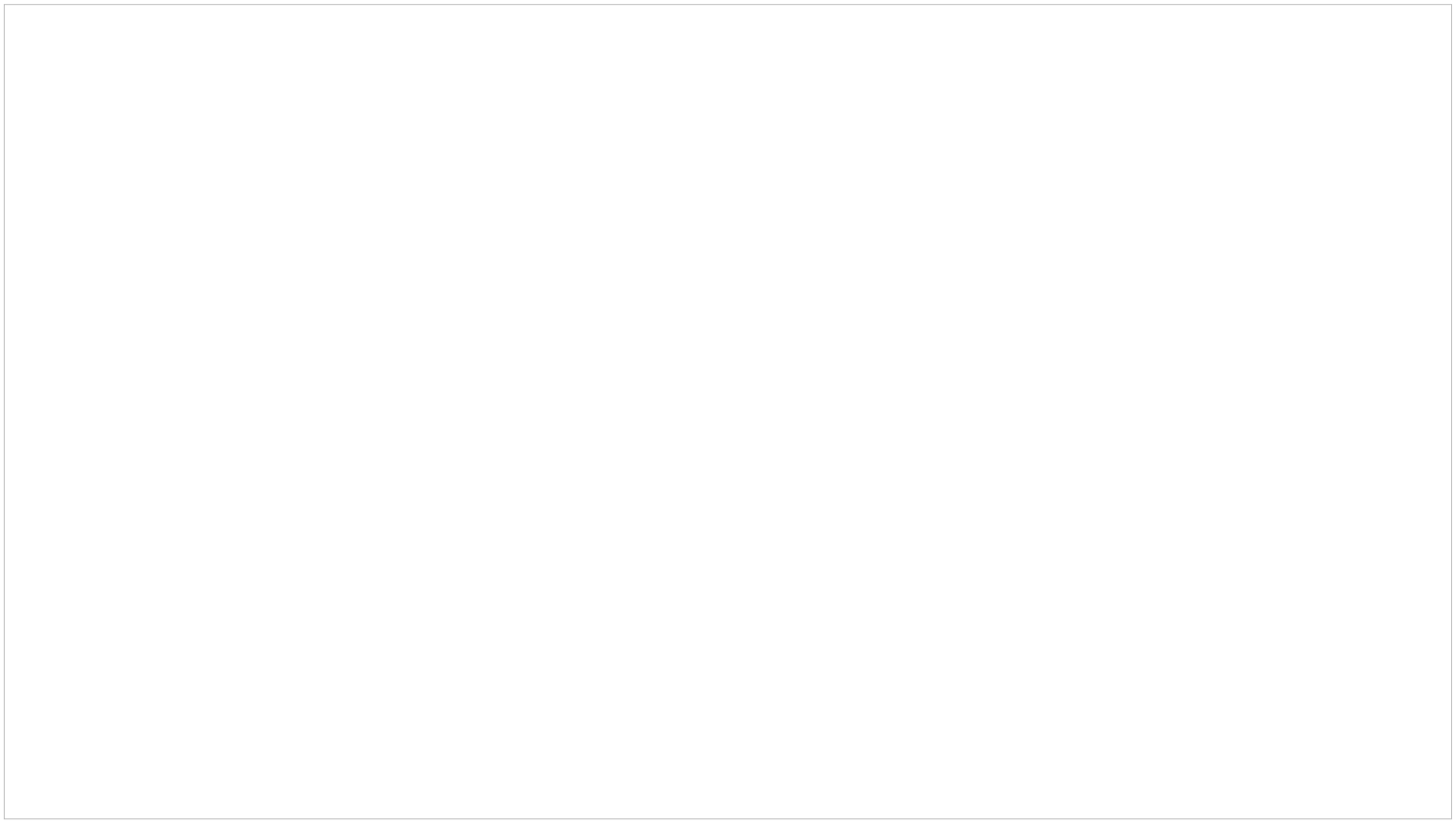
history: interchange already well separated as a separate unix process, communicating via ZMQ.
this comes from early work on GC-like behaviour where the interchange might have been running remotely - talk to Yadu about that. that particular line of development was abandoned but the concepts sort of turned into the GC endpoint.
code example: start_local_interchange_process has the word “local” there because of this distinction.

(mention work to remove multiprocessing fork? and make things either use multiprocessing spawn or explicit processes? not sure if thats relevant to this talk?)

[instruction]

for each language, a code sample, especially highlighting one interesting thing about that language

maybe include one syntax example for each language to make things more concrete - but ideally as part of a concept/lesson learned rather than a standalone syntax example. because syntax is not so interesting as far as this talk goes.



Rust

Modern language targeting the space that traditionally you might use C

Single threaded (maybe real interchange is single threaded now? PR NNNN. If so, this drove some of that thinking / feasibility study). Parsl has been a masterwork of race conditions that I've been spending the last 8 years untangling: concurrency is hard to reason about and to a first approximation should be banned. Which is superficially a funny thing to say when parsl is (from one view a concurrency library) but at the same time, that's also the argument of the task based model, for users: concurrency is hard, here's a simpler safer concurrency model.

ZMQ poll based. Explain poll as blocking call, not a "sit in loop hard looping and checking many conditions every 1ms" - Unix select call introduced in... what year and Unix version?

Strongly typed. Avoid things like dict. This influence is (PR#NNNN) TaskRecord in parsl, which used to be an untyped dict: tradeoff: for prototyping, can dump all kinds of stuff in there. But for maintainability and hackability: tooling cannot tell you if you are trying to use an entry that "doesn't exist" - there is no notion of not existing. Tooling = mypy. Or mouse over in IDE showing documentation. (Screenshot of Python showing a (strongly typed) variable mouseover - if we didn't know the type of this statically, we shouldn't show this mouseover? better to show a variable (where it isn't clear what the object is and needs some type checking, rather than a direct constructor where its extremely obvious what the constructor constructs)

Python does reference counting and garbage collection at runtime. Rust tries to do this statically. Which means you have to reason about who owns objects so that you know when an object is finished with. That can be awkward to get strictly right. But in the Parsl codebase in Python, it still makes sense to think sometimes about who owns an object informally. (Example? For example who is allowed to make calls on a particular object? Rather than chaotically through the source code? Or perhaps that having this kind of formality in general is useful - not so much just ownership)

Rust is used in a bunch of python libraries now: see pydantic-core. (and in PR #3924, the first python 3.14 build error is from pydantic-core not building right because of new versions of python...)

“ownership” model is sometihng you have to think about in other places - even with a garbage collector -- “memory leaks” in a garbage collector model look like keeping. not just about freeing memory, also about what is safe to do at what time.

free memory explicitly-ish (statically?) rather than garbage collector - from the haskell world, see https://pusher.com/blog/latency-working-set-ghc-gc-pick-two/ about GC vs message routers especially. although python reference counting maybe makes that not so bad?

ZMQ monitoring was really helpful - ZMQ is quite async, to the extent that it doesnt tell you when a connection has been opened at the TCP layer. Which can then disguise some errors. It has a monitoring layer which reports progress and I gatewayed that to log messages. I recommend to the remaing core developers to implement this in mainline parsl . Show an example of what messages I got.

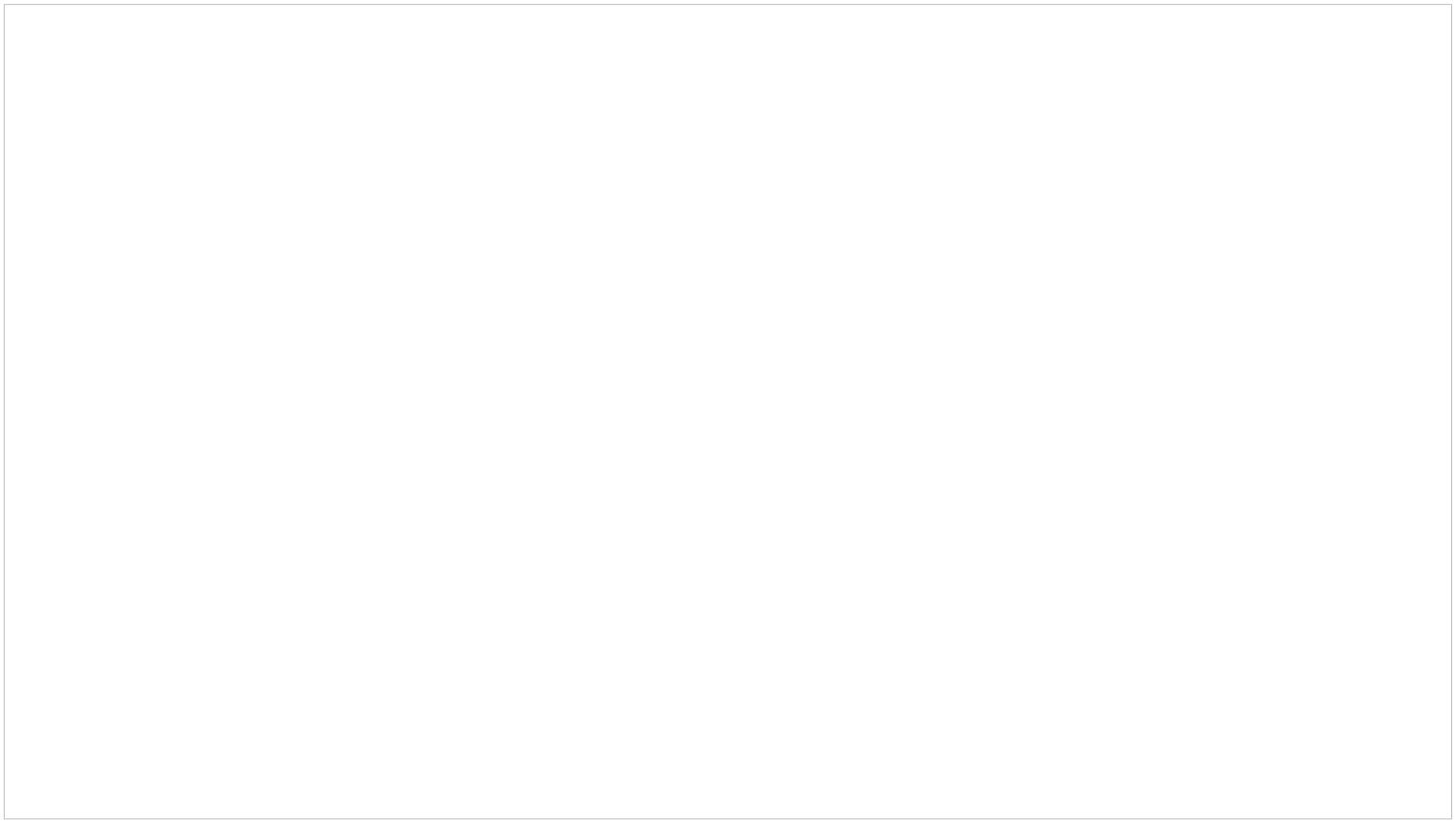
polling not tight looping

(here and in idris2?)

Poll based stuff needs all your pollables to share a pollable representation. In ZMQ poll, can us ZMQ sockets and Unix file descriptors. But in python impl, for example, can't poll on both a ZMQ socket and a multiprocessing queue - which has led to some ugly busy loops in the mainstream impl of Parsl in htex and in monitoring. which uses CPU, and when delays are introduced to avoid that CPU burning, introduces latency on message processing. That can give a recommendation/pressure to choose between concurrency structures.

mentioned radios in monitoring talk earlier

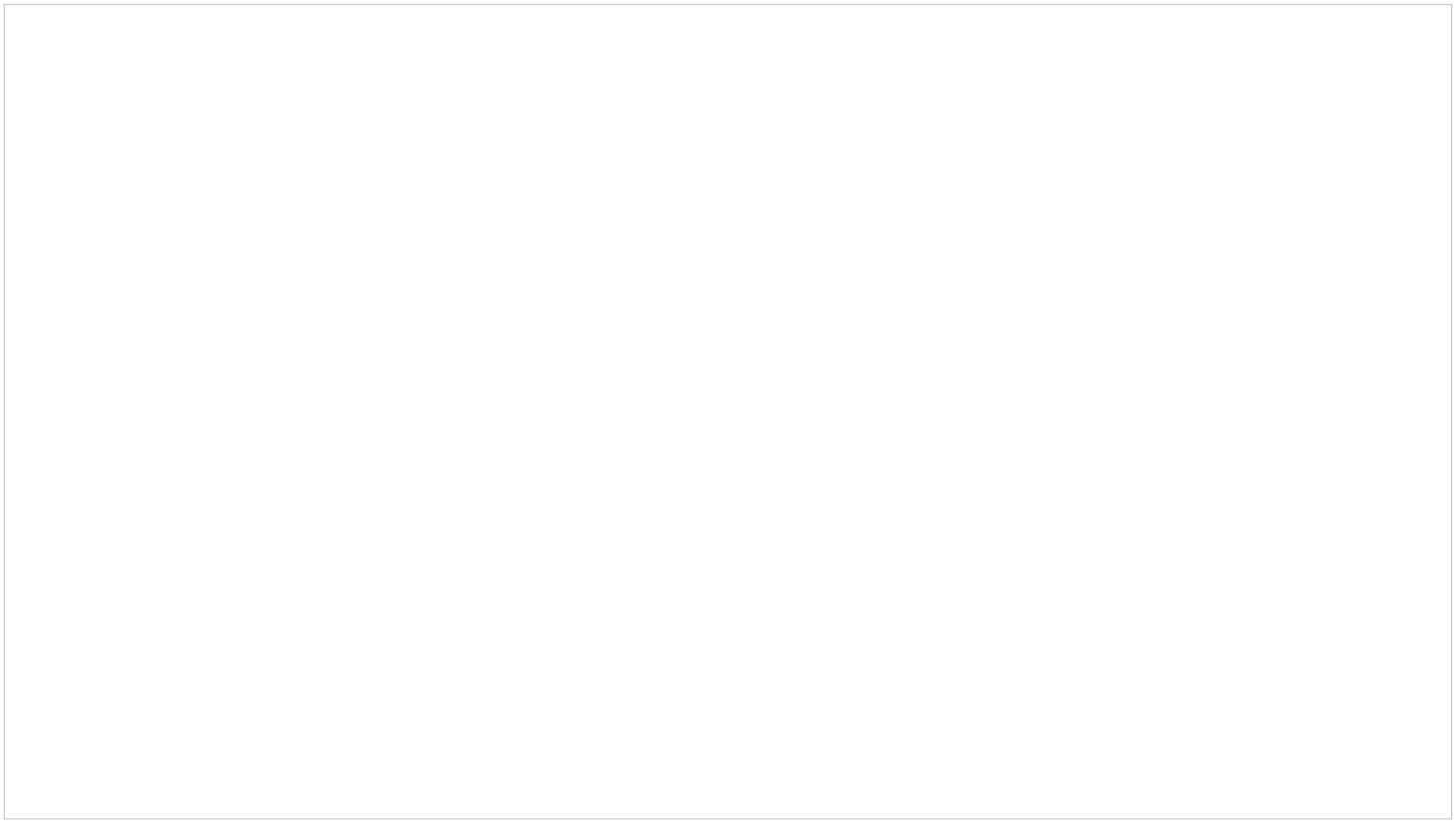
Rust is not so much byetcode stuff as Python, naively assumed it would be faster. it wasn't. put in a slide in this sequence about performance, as a rust artefact.



Elixir
Erlang runtime, various languages on the front: also LFE and Gleam. Ruby inspired syntax.

erlang - telecoms style applications - headline?

Libraries also Erlang. If you are using htex via GC, then your tasks are already passing through an erlang layer - rabbitmq(?)
Contrary to rust impl, this is *many* threaded / what the runtime calls a process, but it is not a unix-level process
Erlang has principle that process should die easily and other stuff should deal with that. One thing parsl has suffered from architecturally is threads and processes dying but then the rest of the pile not detecting that. I've pushed a bit on detecting that in Parsl. But for example it's hard to poll on a multiprocessing queue and also on a process exiting at the same time.
Python threads look like OS threads (which in Linux is very similar to being a Linux level process) - in elixir/erlang, that isn't the case.
Also a principle I've become fond of: you should be eager to raise exceptions but don't try to catch and handle them because you often can't/need to reason very hard about it
Thats their approach to concurrency. Messaging. Processes die easily. Tied processes - if one dies so does a connected process (I have pushed that a bit into Parsl and recommend push more). other shared structures built on that.
htex command client: shares a non-thread-safe structure between threads. can lead to weird behaviour - rare enough that other devs won't fix it. frequent enough I encounter it with my users. erlang runtime puts external interactions on its own process/thread and uses internal messaging to talk to that [beam] process from other [beam] processes more safely. [suggestion - use a cross-thread RPC internally to talk to command client? for example, using ZMQ local connections? it's more bureaucracy but it's thread safer?]
this doesn't eliminiate race conditions - but its a model that can be easier to reason about than Python style shared objects.
one informal effect of moving task launches to own thread (although that wasn't the core intention - was more about stack sizes) but has a more message based model now with queue.
(queues / message queue is the core concurrency primitive in beam) -- PR #NNNN
these BEAM level processes can run across cores and across machines. which means theres a scaling possibility for having a multi-host megainterchange... if the process model was suitably architected. but what is that architecture?
c.f. python asyncio/coroutines: much lighter-weight than Python OS threads (although less capable for multiple CPUs). and yet again a different "poll language".
issue #3761 - interchange should notice when submit process goes away (and vice-versa)
erlang/beam built for serialization - python object model is not (and I have had a lot of fun there ...)
exit handling -- idea that linked processes: if one process dies, the other process should explicity care about it. a little bit manifested in htex heartbeats, but not everywhere - cause of hangs.



ownership (from rust) and thinking about which threads own which things (from elixir/erlang)
(not about who has a *lock* on a structure to prevent concurrent modification, which is a slightly different notion)

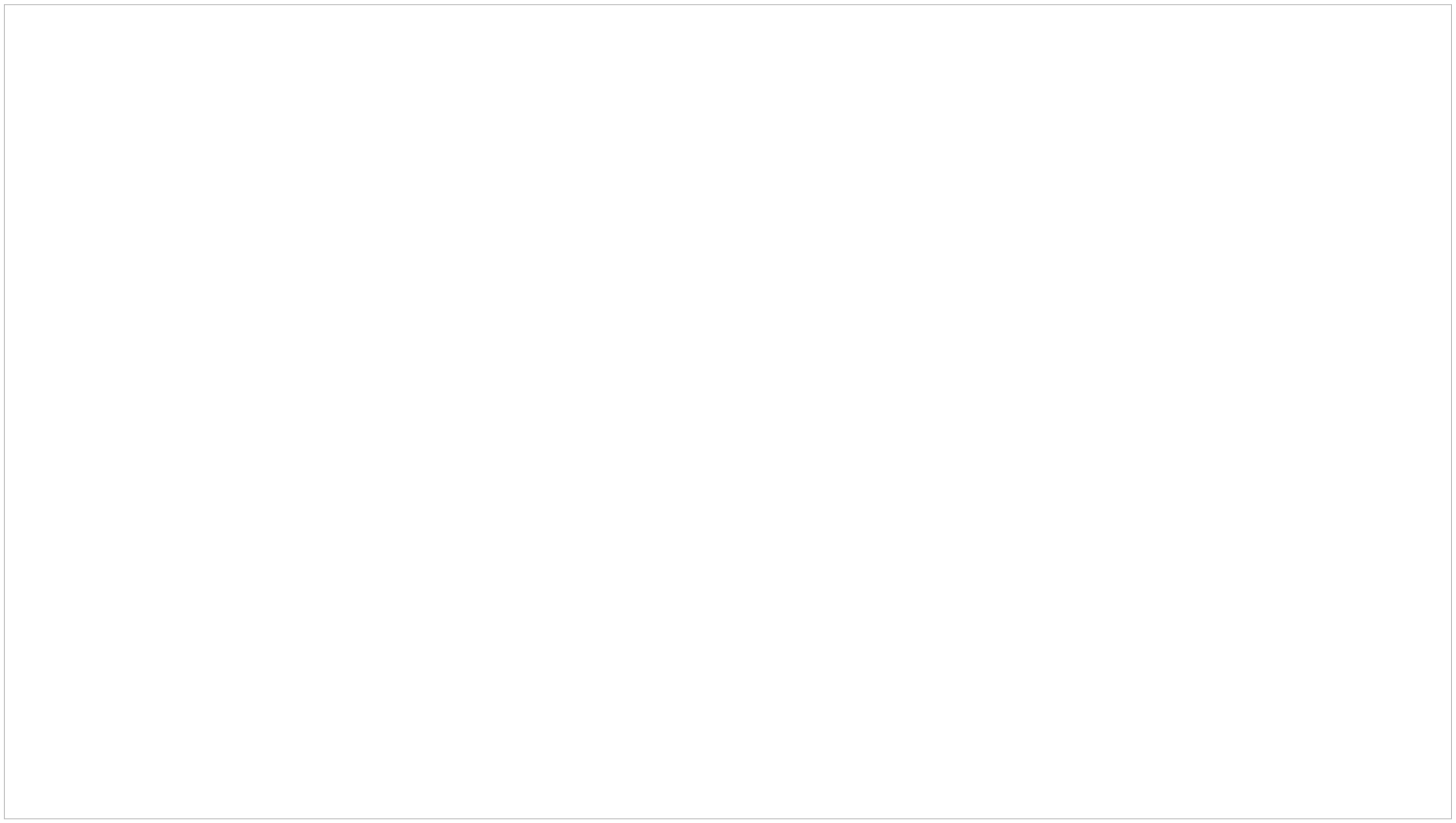
examples of several concurrency problems in Parsl that were made easier by thinking about things in this model:

i) “garbage collection” model introduced in #1512 introduced a whole load of race conditions

ii) htex use of ZMQ: ZMQ is explicitly not thread safe - ZMQ channels must only be accessed by one thread ever (unless you do some subtle non-python-level stuff to move them between threads) - this is an outstanding problem that hits people occasionally with “mystery hangs” and “zmq is not very good” accusations.

iii) more recent (DESC sponsored) shutdown of db manager - occasional problems but recent stuff to do with log reduction exacerbated this I think - two threads in DB manager, but only one of them should be responsible for talking to the database.
(working on a patch, maybe a PR #3922 to fix that)

python’s object style doesn’t particularly discourage you from accessing single-threaded stuff in multiple threads. for example, no type annotations on self.attributes being accessed from multiple threads. which is sometimes fine, but not if those attributes are not thread safe.



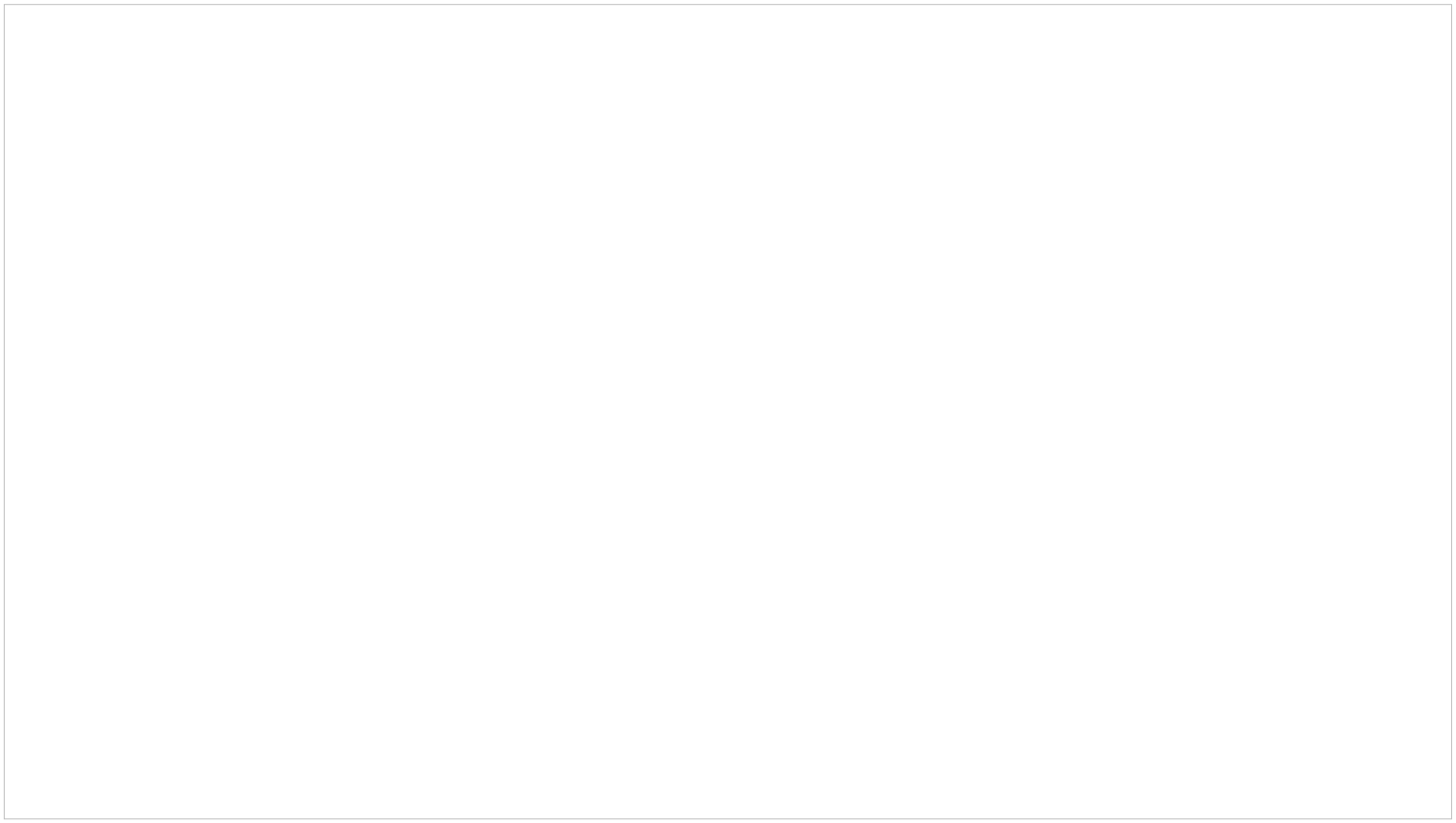
“message mover style”
- lots of bits of parsl move messages around between queue-like structures.

- and have sometimes done other stuff on the way

- which conflates ownership of responsibility

- push towards message mover threads *only* doing moving. not doing other stuff “by the way”.

(db manager shutdown race, example)
(i think some other examples in htex?)

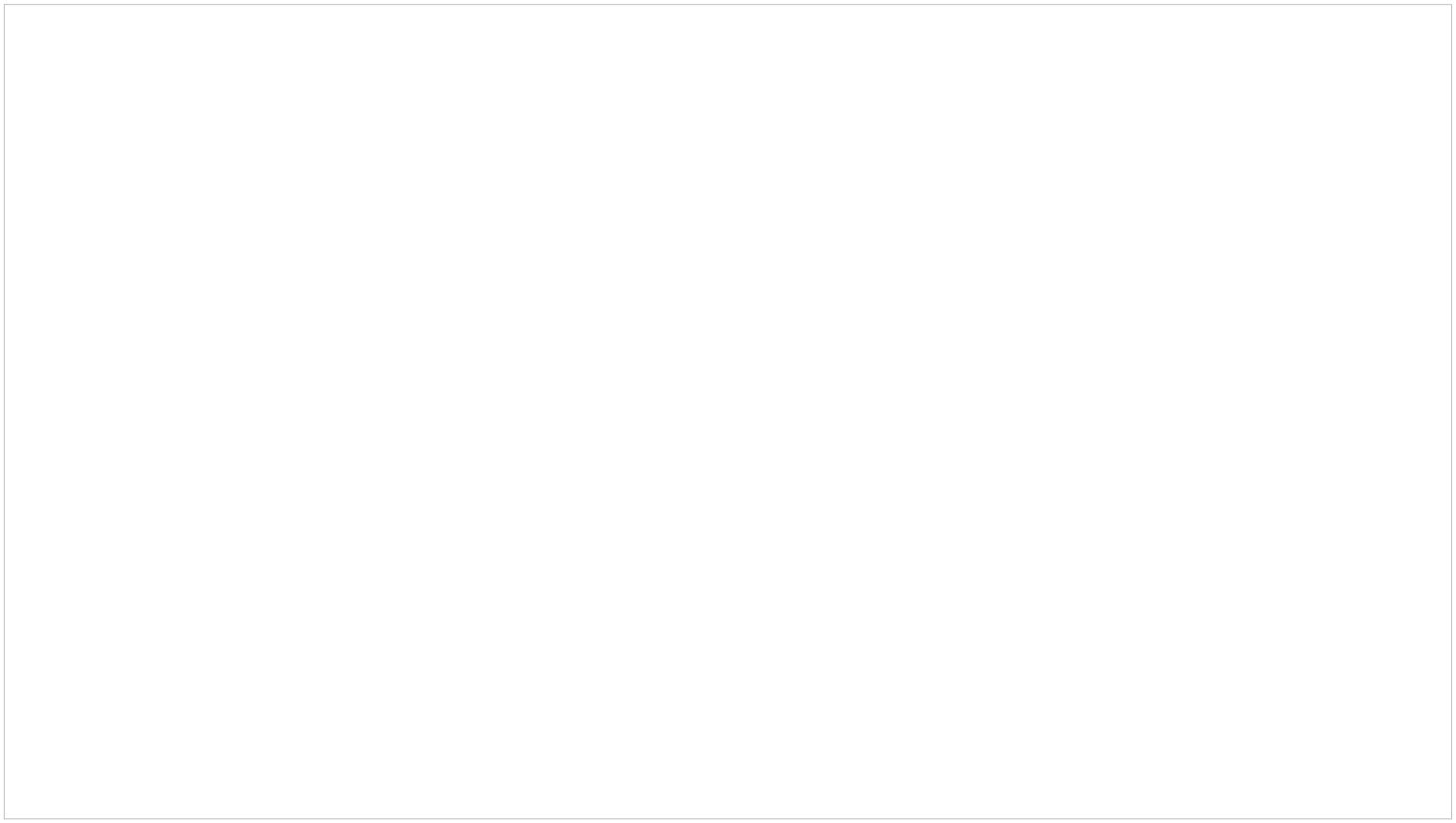


run more threads! (run *many* threads)

parsl monitoring for example had one thread that was both a zmq radio receiver and a udp radio receiver. these were not done in a select/poll way, but making a blocking call to wait, for each. and alternating between the two, using cpu rapidly polling one then the other.

two thread model -- what monitoring looks like now -- we can make a blocking call, and sit there for much longer. we don’t have to alternate between checking two things.

but then, because these two threads are separate, it’s much easier to see that actually we often don’t need the behaviours of one or both, and not start the thread at all -- what looks like “adding threads” actually resulted in removing load.

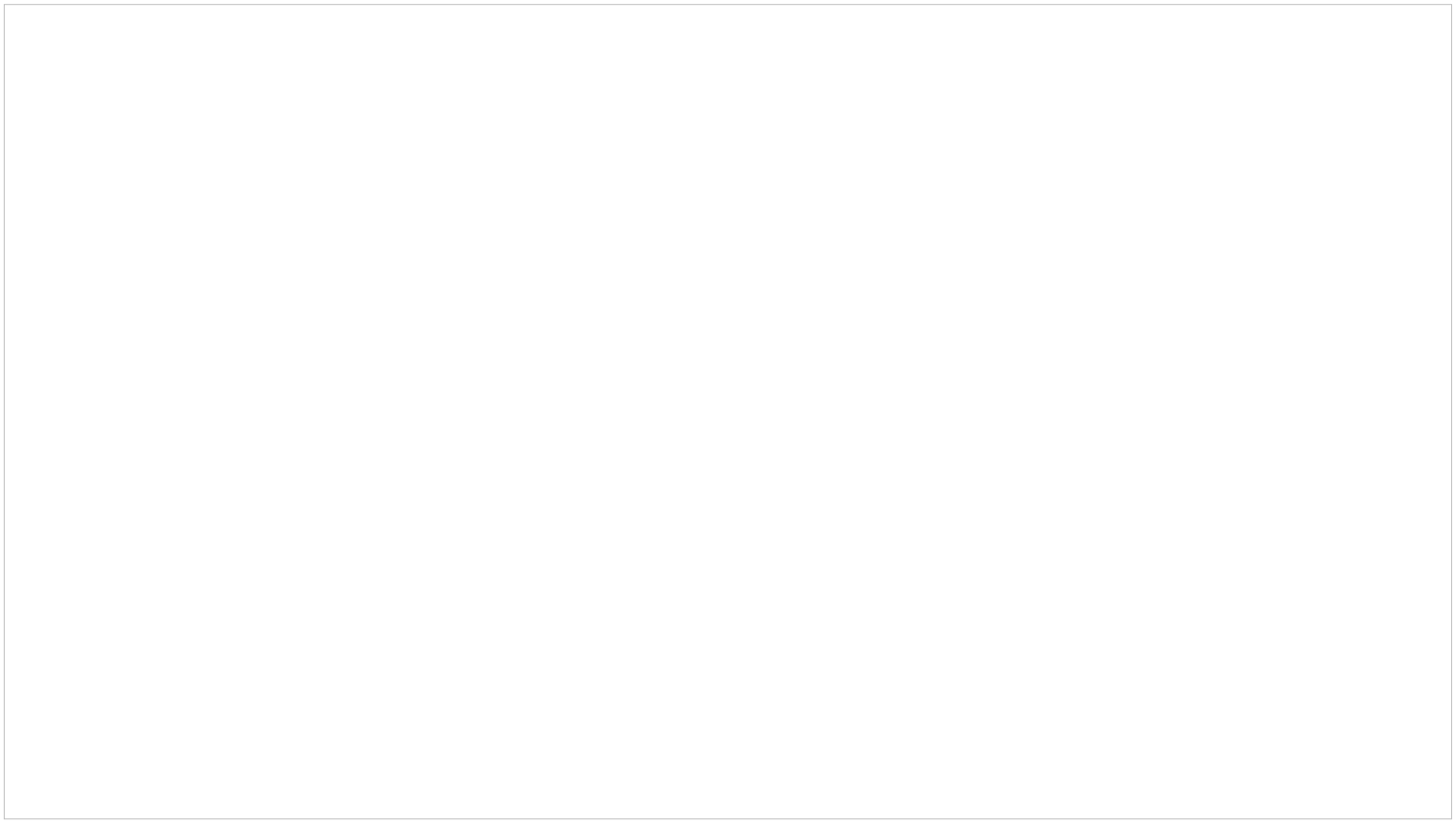


parsl-perf - realised current impl is a bit tricky in its final result when things are non-linear. which turns out to be the case.

perhaps that should be a curve output? to compare with other curves? with doubling task count every iteration up to limit, rather than trying to home in on a limit - nicely gives a tasks/second value, but doesn’t characterise change in behaviour as (for example) number of outstanding tasks changes - which i have observed (or number of workers to dispatch to, for example - which i have not observed)

there’s a parsl-perf talk at parslfest from stefan - talk about that

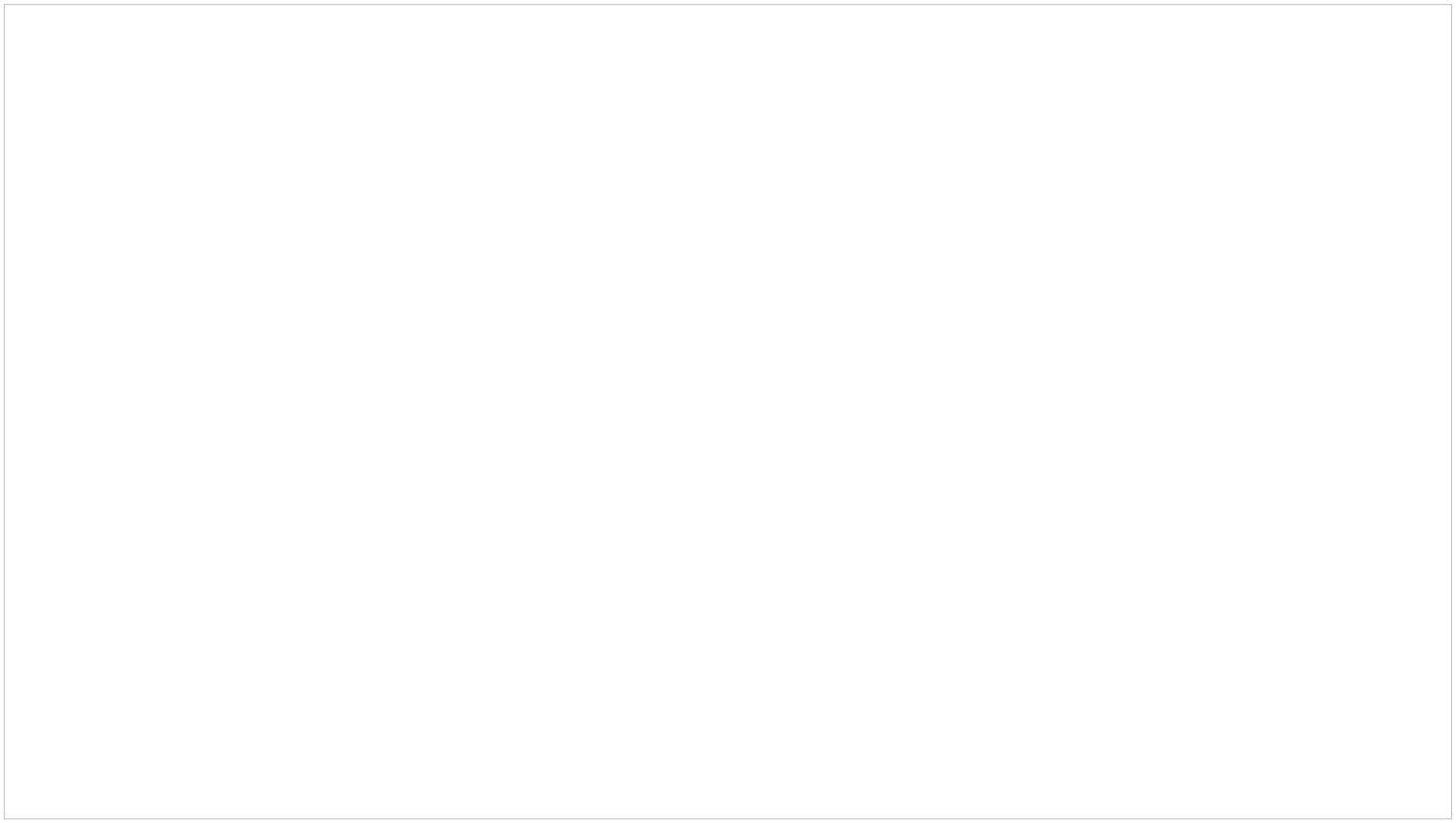
chart of parsl-perf numbers? - with my iterative changes. for idris2 interchange - peak 500-ish tasks per second but drops down to less and can



measuring performance set me off on a whole other project, doing linear modeling of various performance options to try to understand when and how they had effect.

(eg. when is worker prefetch good or bad?)

theres a talk and a writeup [link to all]



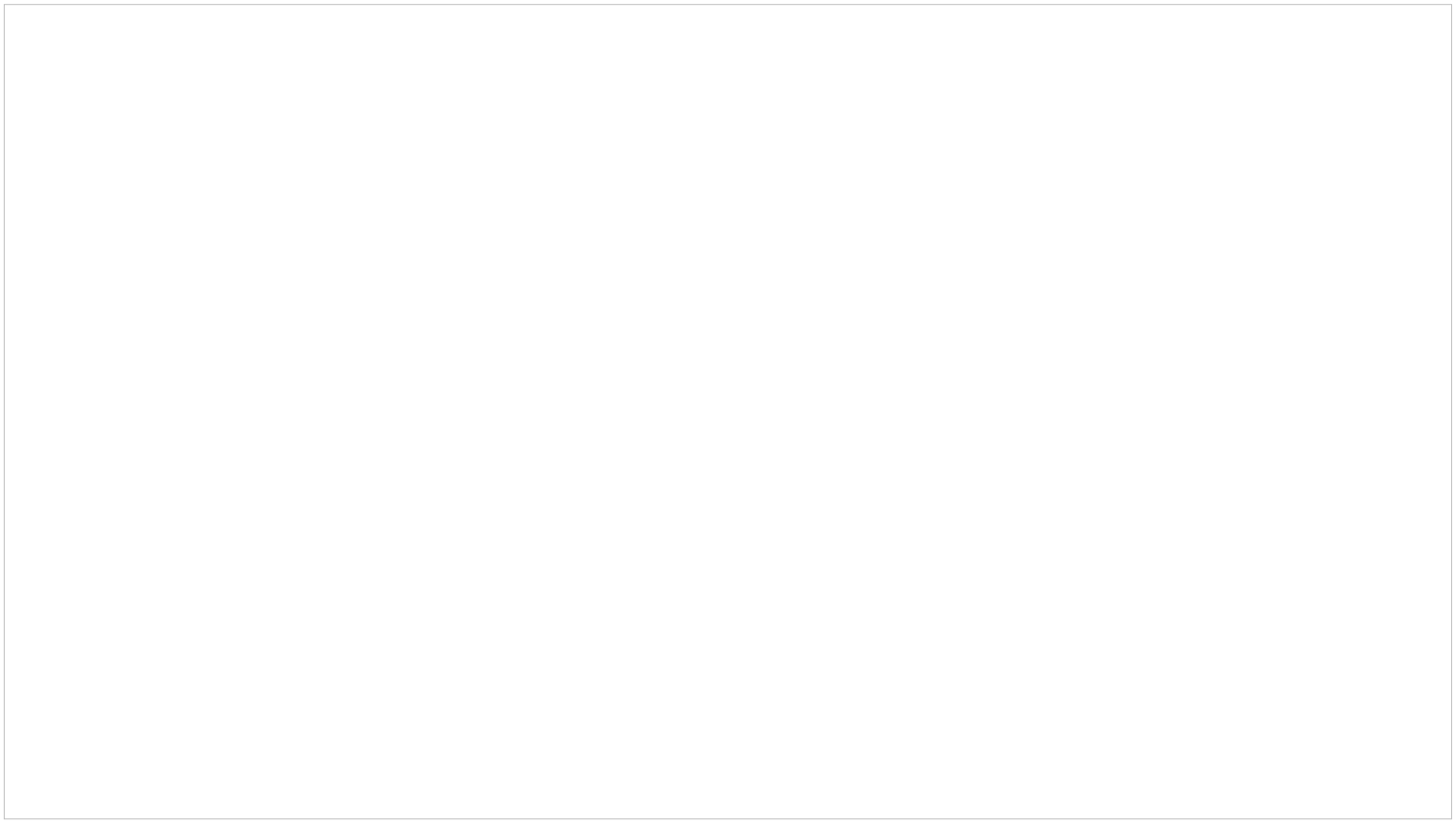
backpressure: noticed this especially with the elixir impl because it scales less well.

but this is a problem everyone has with “many” task workflows. (for some definition of many)
- “just” don’t submit too many tasks to Parsl.

and its part of the motivation for my billion task parsl talk

[link here]

* align this with the performance slide thats at the end



Idris2

headline? purely functional dependantly and linearly typed language [its programming language research] - starting to cross over with proof langauges / theorem provers.

Runtime on top of scheme. Syntax is very much Haskell but different - which standard way to do things (and Haskell was designed to “serve as a basis for future research” - [influence box: that’s part of what leads to my thinking of Parsl being a hackable pluggable core on which people build their own particular chaos] - many language features in mainstream languages developed in the likes of Haskell

Much more experimental language. So a lot rougher to use. (eg. error message quality - something Rust people have put a lot of work into; package management is rough; and roughnesses that maybe compiler bugs or unimplemented compiler functionality - everyone here is used to Research Code) and i have slighly modified the compiler stdlib. (although I’ve also done that for my own Python installs...)

UNIX side track - not what I was planning, but got distracted having a nice look at kernel stuff which I hadn't looked at for a long timeUNIX poll based. ZMQ exposes a Unix FD for the purpose of using with unix poll. Also novel poll types (I think pidfd? Or signal FD? Or both?) - pidfd lets us poll on other processes ending. So interchange ends when the submitter ends. (Did I implement this parent-pid in mainline parsl? If so, pr #NNNN - but looping poll not poll based poll) -- pidfd for doing that Erlang style “if that other process goes away, this process should too”. Traditional parsl bugs of processes not going away.

poll, ioctl, special opens, special message formats - these are not “files”

TODO: can I do a bit of phantom typing on the FD to give me pollable? In progress now, maybe another compiler bug. But it’s a nice “transcribe kernel rules into the type system, rather than using ints”.

Another slide - quick snapshot of other interesting things that can be represented as fds. FD abstracts a "kernel object" not a file. (My preferred phrasing to "everything is a file" that Unix people like) eg poll for a raspberry pi IO pin change.

So modelling event loop as fd based.

Single threaded.

Strongly typed like rust...But even more so.

Static checking resource management - not just that we want to ensure that free is called.

Polling thoughts here really made me think about common model for polling otherwise stuff doesn't work.

I used pidfd (signals for other reasons aren’t delivered to the right process but I would like to catch signals via fd too)

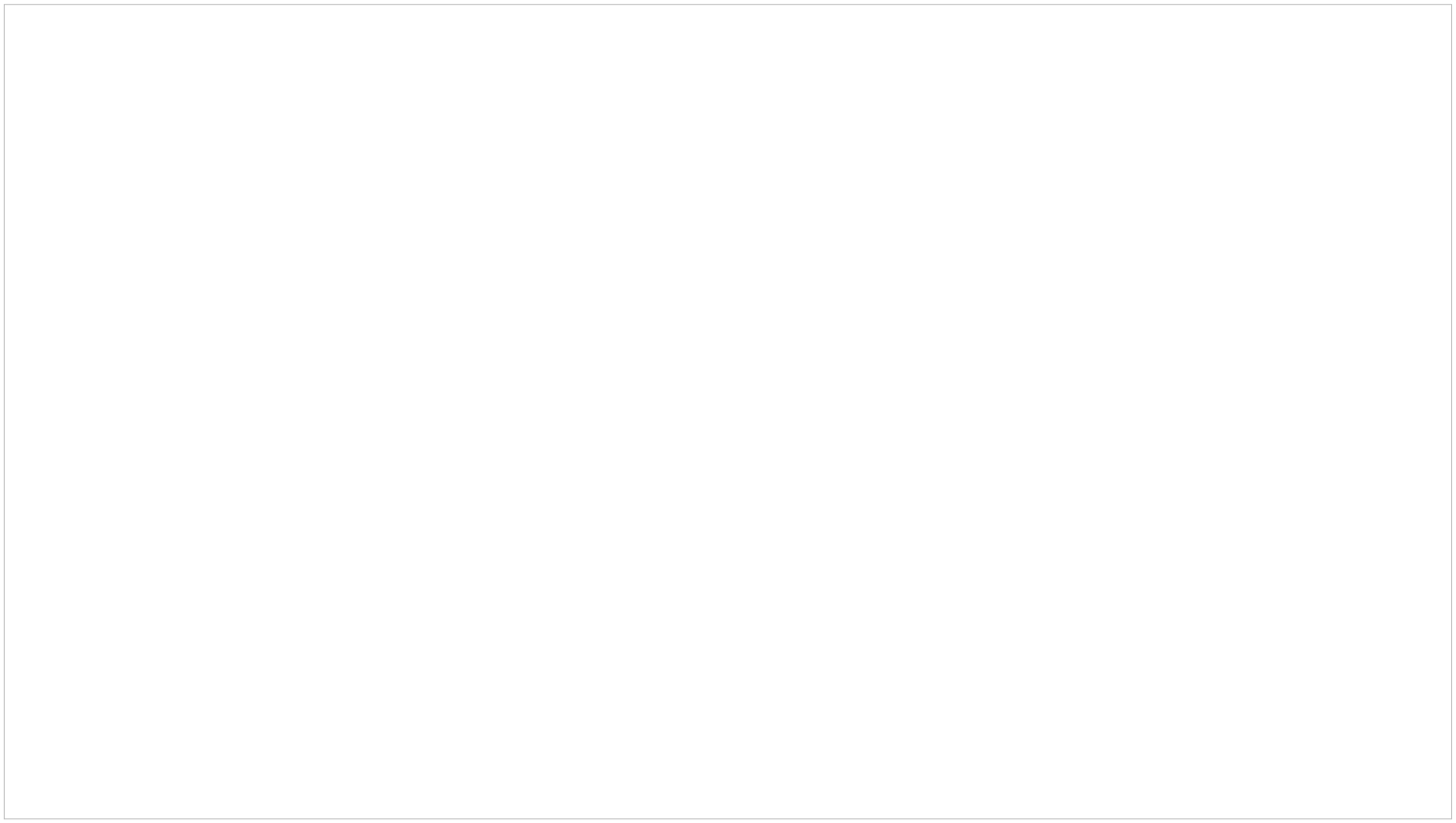
Classic example of dependent types is reasoning about vectors: like a list but type safe around length.

Example of linear types is resources rather than values: eg. memory buffer

built my own lower level libraries - pickle imp, zmq C binding, unix/file descriptor interfacing, bytes, logging - show sizes

linear types: can check a value is used exactly once. quite similar to rust’s ownership and implicit frees when no longer needed. but might imagine things like: “you have to do *something* with this task - don’t implicitly free it, because implicitly freeing it is memory-safe but it’s not higher level safe: we should *never* silently discard a task. worst case would be to log a critical error when doing so. but other thing we can do is dispatch it to a worker and then insist we deal with a result? (which also involves some wire protocol). Maybe i can do some stuff around there in the next few weeks?

(“implicit” frees can be bad because you don’t necessarily know when they’ll happen: plenty of parsl hangs, especially around exit, have come from the __del__ method of objects; which fires in an arbitrary thread at an arbitrary time. this works in parsl especially badly with ZMQ)

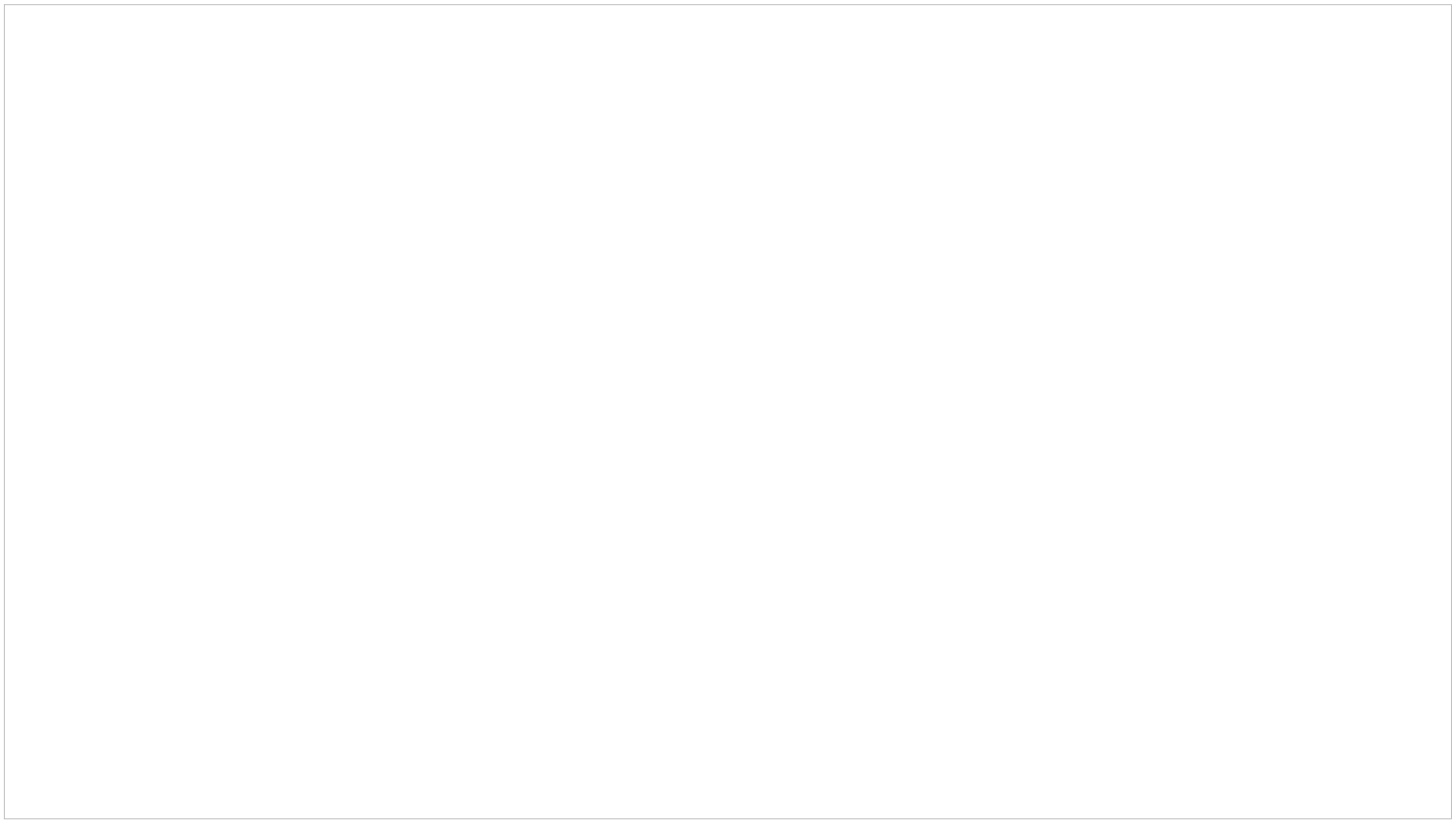


case style (see match syntax introduced in python 3.10 that can be used in Parsl from next month...)
- I think rust has this too, not sure how much it does exhaustiveness checking?

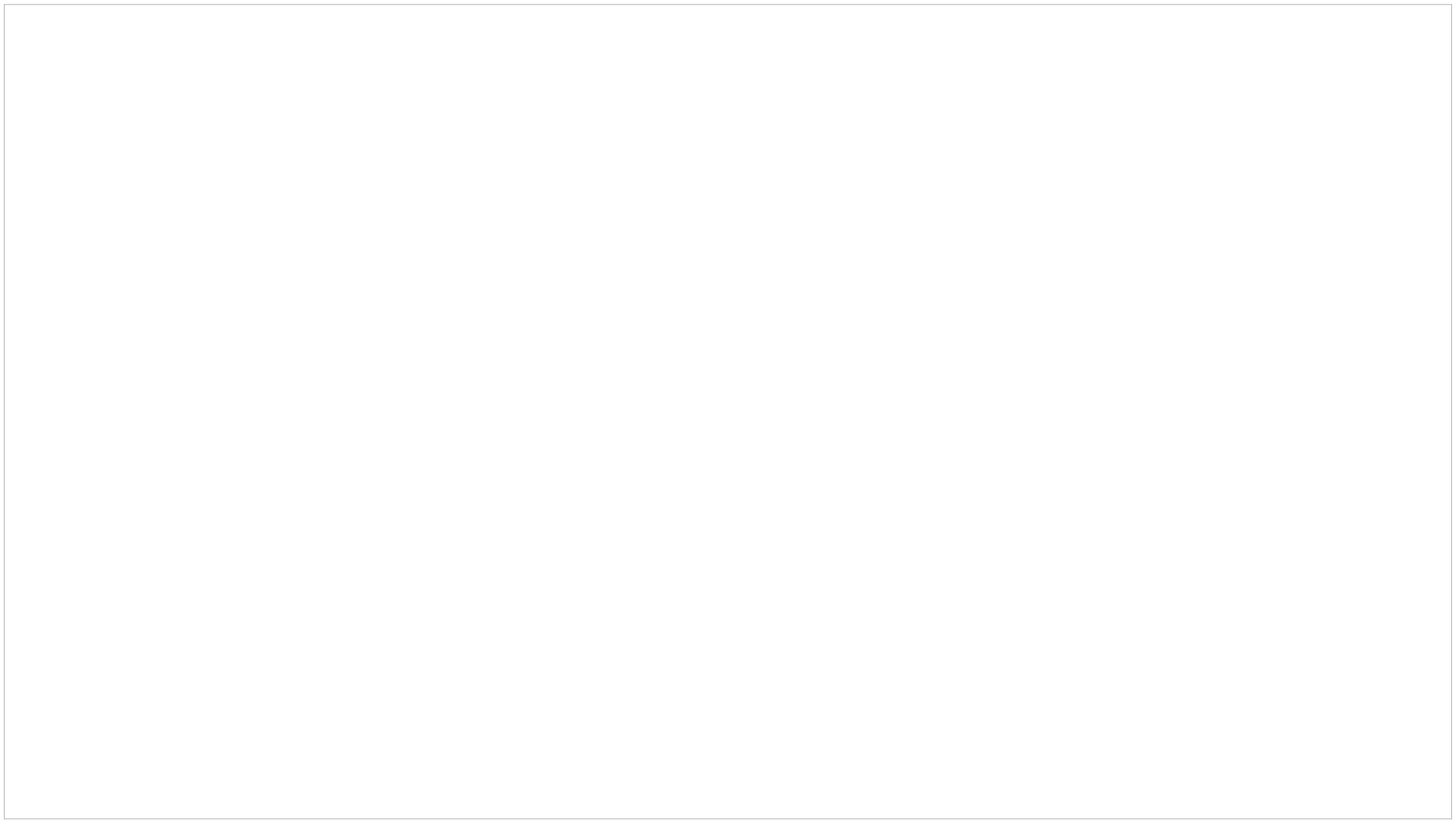
strong exhaustiveness checking of that pushes a lot on “what to to do when something doesn’t match”. it isn’t super clear what the right / tested behaviour is in the interchange - 34 cases in idris2interchange where we just abort the whole interchange (which causes Parsl to hang -- see issue #NNNN about interchange going away).

mostly in Pickle deserialization, triggered by malformed pickle messages.

- my personal style: “exhaustiveness checking” has led a review style: “why doesn’t this `if` have an `else`?” that catches lots of naive issues. (the `if` made an exception not happen right there, but didn’t actually address the behaviour when it didn’t fire)



didn’t really get much done with linear types. which was the point of using idris2. oh well.



Testing: test suite isn't set up for external process interchange very much - not too surprising/something to blame parsl for. But in future, if GC pushes Parsl towards putting different versions together, it's probably interesting to push testing towards pulling different pieces from different parsl codebases and trying to make them work together - again principle that "older parsl were testing against might be arbitrary code in arbitrary language" - not "well it's probably really the same as the version we're testing now"

not testing eg: monitoring message flow (see other talk). priority queue. lost workers. sending the right number of tasks to a worker. (not testing almost anything off the basic path)

example: issue #3022/PR #3871 incompatibly changes the worker/interchange protocol. so I stopped merging master branch at this point. but people forever mumbling about using different versions of parsl together (eg. building a container with workers and expecting it to work across parsl versions).

maybe not so relevcant submit<->interchange, but is relevant interchange<->workers

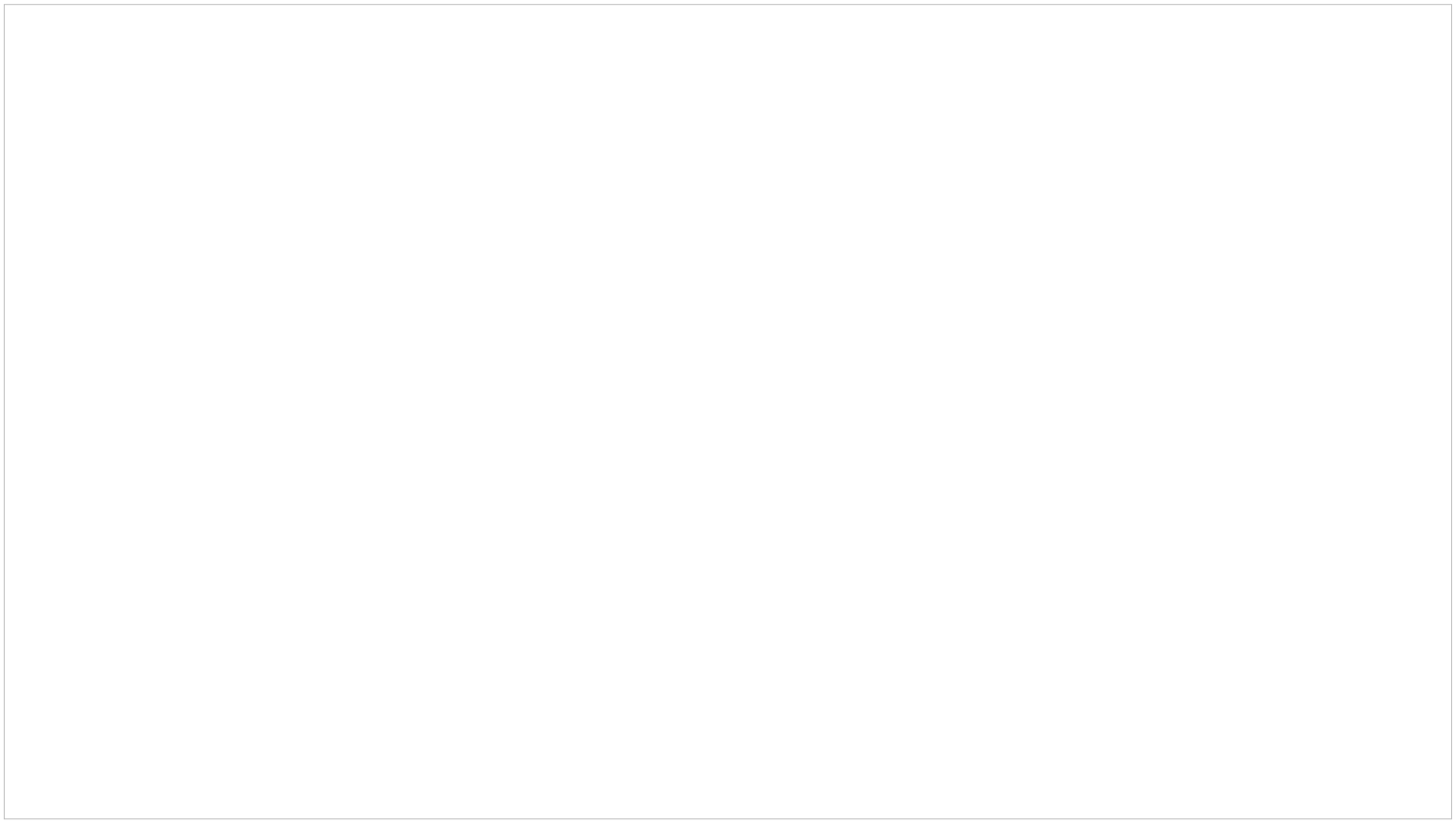


Diagram: source code sizes, on stack

Mention scaling and parslperf results. And two follow on / adjacent talks that grew out of this: billion task parsl (by billion, I'm just picking an integer value for infinity) and linear model approximation of scaling parameters
- how interchange didn't seem to change speed, so i ended up down a rathole wondering what else would change rate? aka on my laptop, python interchange was not a rate-limiter.

(spoiler: it's logging)
(example, 15 t/s idris2 logging on, 400 t/s logging off)
(also give numbers for htex, init_logging flag)

billion task parsl: https://www.youtube.com/watch?v=5brIeAvZG1c

performance analysis of other things: http://www.hawaga.org.uk/ben/tech/parsl-r-perf/