Wide event observability prototype report

Ben Clifford

Dec 03, 2025

CONTENTS

CHAPTER ONE

INTRODUCTION

These are notes about my current iteration of a Parsl and Academy *observability* prototype. It is intended to help with plugin style integration between those two components and an open collection of friends including Globus Compute, Diaspora and Chronolog.

As an abstract concept: "Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs." (https://en.wikipedia.org/wiki/Observability). In the context of this project, it means outputting enough information about the system to understand why bugs happen in the system.

There is plenty to read about Observability on the web: google around for more info.

This is often neglected as part of the core functionality of a research prototype, as demonstrations are run in controlled environments with the original authors both ready to respond to the slightest problem with copious time, and ready to restart everything from scratch repeatedly until the desired outcome is achieved.

As soon as that prototype is forced into production, those two properties evaporate and the need for observability manifests: both users and legacy developers need to understand what is happening in this suddenly wider and more hostile world.

In the Parsl world that exists now as two separate systems: Log files and Parsl Monitoring. This report will explore ways in which they can be usefully unified and extended.

This project builds on experiences debugging Parsl within the DESC project, as well as work sponsored by NSF and CZI understanding pluggability and code maturity as they affect architectural decisions. As a concurrent activiy, I have used some of this experience to push changes into the Academy codebase to support its move towards production.

A distant vision is a project-wide or personal-space-wide observability system – but it is important to acknowledge that this is a distant and vague vision, and that actually what I want to happen is stuff on the scale of weeks to months that is usable on that timescale, with others left to take up that distant vision if desired.

This report attempts to describe abstract concepts but ground them in practice and concrete code-driven examples. This report also tries to give open questions and opportunities that might be interesting for other people to work on.

How to try this out? Because I want you to try this out. It's in my Parsl `benc-observability` branch. I will try to label use cases as expected to work or not, and in what context. There will also be some academy-related stuff in that branch, with the intention that it moves elsewhere as productionised.

1.1 What exists in Parsl now?

Parsl has two observability approaches: file-based logging and Parsl Monitoring.

File based logging is very loosely structured. Log lines are intended for direct human consumption, with minimal automated processing: for example, "grepping the logs". Within Parsl there is a variety of log formats, usually depending on the component which generated the log. Logs are directed to a filesystem accessible by the particular component, which in practice they are especially awkward without a shared file system. It is easy to add a new log line, by writing what is effectively a glorified `print` statement.

Parsl Monitoring generates monitoring events that are not intended to be seen by humans. Instead they are conveyed to a Monitoring Database Manager which munges them into a relational schema. This schema is then typically accessed by users via futher processing for pre-prepared visualization or ad-hoc queries designed by data-science aware power users. These monitoring events can be conveyed by a pluggable interface, for example over the network, and in contrast to the logging approach above, distributed filesystem access (in the broadest sense) is not required. The strict SQL schema makes the data model extremely hard to extend ad-hoc.

Parsl Monitoring was also implemented with a fixed queries / dashboard mindset: one set of views that is expected to be sufficient. As time has shown, people like to make other outputs from this data.

This report builds on both of these approaches. I'll talk about more details in later sections.

1.2 Diagram

of the components/flow.

to distinguish the pieces of my work, and also to distinguish the pieces of what might be substituted where.

specific emphasis that this is common techniques, not a single implementation or protocol standards or single anything.

```
Python logger
  API    ----> JSON structured logs \
                                    |--> log␣
↪movement --> Python-based query model            ␣
↪--> graphs/reports
        non-JSON structured logs   /    to one␣
↪place     --> post facto schema normalisation
```

(continues on next page)

(continued from previous page)

```
        (eg. WQ, Parsl monitoring)  classically␣
↪files,          --> data-structure based queries
                                    but eg ryan/
↪kafka
                                           logan␣
↪demo/agent polling
```

1.3 Concept: Universal personal logging

Imagine *all* of my academy/globus endpoint/parsl/application runs going into a single log space. permanently. no matter what the project, location, etc.

heres a logging use case/notion: universal personal log. all my GC endpoints, parsl runs, academy runs, application submits, go into a single log space that has everything I am running everywhere in the science cloud, by default - eg. identified by my globus credential ID. no separation whatsoever. no project distinction, etc.

what does that look like to work with on the query side. what does that look like to query?

1.4 Target audience

This project is mainly aimed at systems integrators and application builders who are expecting to perform serious debugging and profiling work at a deep technical level. It should support other use-cases such as management-friendly dashboards.

These users will often have integrated several research-quality projects: for example, Academy submitting into Globus Compute. As systems integrators and application builders, they aren't directly interested in the borders these individual projects have built around themselves, but want to understand (for example) where their simulation task is inside the whole ad-hoc stack. This mirrors the microcosm of Parsl existing as a pile of configurable and pluggable compponents, each with their own observability options.

Many of the target audience do not, in my experience, come asking directly for observability as a feature. Instead they come with questions such as “Parsl is slow - how can I make it faster?”. Without understanding the statement “Parsl is slow” (which as often as not turns out to be “my application code is slow”), it is hard to make progress on “how can I make it faster?”

1.5 Modularity

This report emphasises modularity as a core tenet, to the extent that a single product codebase is not particularly an end goal.

1.5.1 Modularity as a requirement for a rich research landscape

A “rich research landscape” means many components, each with competing priorities for productionworthiness vs research. TODO: cite the work done as part of NSF/CZI sustainability grants about recognising the difference between goals rather than ignoring them.

Expecting a single observability system to provide all needs is unlikely to succeed in such a varied research-style environment: while some users are tolerant of appalling code quality in exchange for interesting research results, those same users require production quality from other components in the same stack; and those tolerances vary with every use case.

1.5.2 Hourglass model with several waists

the hourglass model is intended to provide a small number of plugin/intergration points in the same way that the Internet Protocol does for applications/application protocols vs networking technologies (for example, HTTP over mobile phone network vs telnet over ARPANET is then sufficient integration to run without more work: telnet over mobile phone, HTTP over arpanet)

The hourglass waists are:

- python `logging` system: any Python code can send log messages to the built-in `logging` system and any Python code can register to receive any log messages. This can support components that live in the Python ecosystem. That includes enough of the current ecosystem to consider specially, but not enough to be universal: for example, when running task through Parsl's Work Queue executor, a substantial piece of execution happens in code written in the C programming language.
- JSON records: a second point of modularity is representing observability information as JSON objects. This is flexible data format which complements the Python code approach of the previous waist. Often observability information which cannot flow through the Python `logging` API can flow as JSON records.

1.6 High level structure of this project

This report breaks observability into four rough parts:

- A data model of wide event records: *The data model*
- Creating wide records: *Generating wide records*
- Moving those event records around: *Moving wide records around*
- Analysing those records: *Analysing wide records*

CHAPTER

TWO

THE DATA MODEL

2.1 Introduction to wide events

as JSON objects, as Python LogRecords, as roughly isomorphic structures

wide in the style of denormalised data warehouses, rather than heavily normalised like more traditional relational model.

they should be wide and flat: do not create elaborate object graphs. key/value, with values being simple.

is often ad-hoc: people are writing code to run tasks, not building data models represent the observable state of their tasks. so don't bake that into the system too much, and expect to be flexible.

2.2 What exists now: Parsl python logs vs Parsl monitoring records

Especially for this chapter, how both of those can be embedded as wide events

Parsl monitoring structured records are equally valid examples of existing structured records, alongside with equal value to logging as differently structured records.

Original Parsl monitoring prototype was focused on what is happening with Parsl user level concepts: tasks, blocks for example as they move through simple states. Anything deeper is part of the idea of "Parsl makes it so you don't have to think about anything happening inside". Which is not how things are in reality: neither for code reliabilty or for performance.

often want to debug/profile whats happening *inside parsl* rather than *inside the user workflow* - and the distinction between the two is often unclear.

2.3 Optional and missing data in observability

log levels - INFO vs DEBUG

missing log files - eg. start with parsl.log, add in more files for more detail

Security - not so much the Parsl core use case, but eg GC executor vs GC endpoint logs vs GC central services have different security properties. Or in Academy, the hosted HTTP exchange.

The observability approach needs to accomodate that, for any/all reasons, some events won't be there. There can't be a "complete set of events" to complain about being incomplete.

less data, well the reports in whatever form are less informative, to the extent that the lack of data makes them so.

This optionality aligns with different components adding their own logs, if they happen to be there.

Adding (or removing) a log field is a lightweight operation

2.4 Data types

data types don't matter much for human observation. but for machine processing they do. so this section has some relevance when thinking about the Analysis section later.

Both JSON and Python representation can support a range of types. But richer data types can exist. What's interesting for analysis is primarily relations like equality and ordering, which are implicitly not string-like in a lot of cases. For example:

- string vs int: '0' vs 0 as a Parsl task ID. Even Parsl source code is not entirely clear on when a task ID is a string and when it is an int, I think. Normalisation example: "03" vs "3". Rundir IDs are usually at least 3 digits long. Is rundir "000" the same as rundir 0?
- UUID as 128 bit number, vs UUID as a case-sensitive/padding-sensitive ASCII/UTF-8 string. uuids should be used *more* in this work - they were invented for the purpose of this kind of distributed identification https://en.wikipedia.org/wiki/Universally_unique_identifier#History https://www.rfc-editor.org/rfc/rfc9562.html Normalisation example of string form: case differences.
- ordinal relations of text-named log levels (WARN, WARNING, INFO, ERROR, ...) in various enumerations (although for querying an overarching schema is maybe possible for read-only ordering use)

Whats the right canonicalisation attitude here? open question for me.

Cannot expect emitters to conform to some defined canonical form.

perhaps I should use rewrite_by_lambda(logs, keyname, lambda) to change known fields into a suitable object representation: int for some task IDs, UUIDs, log levels? then the type system deals with it? some of those could happen in the importers, some in the query, as its modular. I already do a rewrite to shift the created time down to 0 base, in one of my plots. So the notion is there already.

2.5 Distributed state machines - parsl issue #4021

“distributed” state machines are hard. see parsl-visualize bug #4021 for an example.

don’t try to make the data model force this. the events happen when they happen. handle it on the query/processing side: you make whatever sense of it that you can. it’s not the job of the recording side to force a model that isn’t true.

for example, in the context of #4021, we might want to project some external opinion that running should “override” launched for a task, that isn’t reflected in the emitting code/emitting event model at all, based on an artifical “force to single thread” concept of task execution. in the same vein, the #4021 suspect tasks have *negative* time in launched state. which sounds very weird for a non-distributed state machine model.

or we might only want to visualize the running/end of running times and forget overlaying any other state model onto things: the running and running_ended times should at least be consistent wrt each other as they happen to come from a single threaded bit of code.

see also that the parsl TaskRecord never records a state of running or running_ended. despite it being a valid state. this already only exists elsewhere in the system as a reconstructed state machine - not a real single-threaded/non-distributed state machine. so parsl monitoring is already a demonstration of the violation of this model.

2.6 commercial observability vendors

honeycomb, or built into AWS as cloudwatch

more ad-hoc construction, less buy in from components, rather than all working together to build a single platform, which is often how the commercial observability usecases are described.

OpenTelemetry as a standard. How does this related to that standard?

TODO: maybe opentelemetry is better in *Moving wide records around*

2.7 The argument for templating log messages

previous argument: avoids string interpolation if message will be discarded.

new argument: we can use the template to find "the same" log message even when its interpolations vary.

advanced level question: when a task changes state, should the template be interpolated or not with the state names? because in my example query, it is relevant to see those changes *not* as templated away.

2.8 Objects and spans

it is a "thing we want to talk about", as a very weak notion.

weak notion of "object" but it does exist: for example, logs that are about a particular Parsl task, or a particular HTEX worker, or a particular batch job.

multi-attribute keys - sometimes hierarchical but that isn't required.

eg. contrasting parsl task IDs vs parsl checkpoint IDs: in parsl checkpoint world, tasks are identified by their hashsum. there might be many tasks that run to compute that result. when working cross-dfk checkpointing, the cross-dfk sort-of-task ID is that hash sum, in the sense of correlating tasks that are elided due to memoization with their original exec_done task in another run. (so hashsum is one form of task ID, parsl_dfk/parsl_task_id is another - both legitimate but both different)

cross-ref "span" concept from other places in more broad Observability.

Compare with https://en.wikipedia.org/wiki/Entity_component_system which has identified entities, where the entity only has identity and no other substance, along with components as "characterising an entity as having a partcular aspect" and the "system" which deals with all the entities having particular components. This fits the partial data model fairly well, I think: the notion of identifying entities, without those entities having any further structure; and then the data you might expect to find about certain entities being orthogonal to that.

2.9 Entity keys

This is probably the fundamental problem of JOIN here, compared to traditional observability which passes request IDs around up front.

In a traditional distributed object model system, you would use something like UUIDs everywhere. However, this observability work is not observing a traditional distributed object model system.

note that in parsl some IDs are deliberately not known across the system at runtime because it would be expensive to correlate them in realtime, and that is not necessary for the executing-tasks part of Parsl, even though its necessary for the understanding-how-that-task-was-executed section.

2.10 Other components

rewrite this to not be parsl-centric but instead talk about integrating "objects" from different components even though those components are not strongly aware of each other. wq vs parsl task id is a nice example: regularly used, but log files are different formats, identifier space is different, cardinality of tasks is different: one parsl task != one wq task.

Some components aren't Parsl-aware: for example work queue has no notion of a Parsl task ID. and it runs its own logging system, that is not Python, and so not amenable to Python monitoring radios.

ZMQ generates log messages which have been useful sometimes and these could be gatewayed into observability.

inherently chaotic research prototypes can benefit from observability - as part of building and debugging them, rather than a post-completion 2nd generation feature - but that is impeded by requiring a strict sql-like data model to exist, when the research prototype is not ready for that. (see attitude that monitoring is something aimed at “users” later on, not something that is aimed at “developers” understanding the behaviour of what they have created)

“user” applications adding their own events, and expecting those events to be correlatable with everything else that is happening, is part of the model: just as we might expect Globus Compute endpoint logs to be correlatable with parsl htex logs, even though Globus Compute is a “mere” user of Parsl, not a “real part” of Parsl.

should be easy to add other events - the core observability model shouldn’t be prescriptive about what events exist, what they look like. even though someone needs to know what their structure is.

to that end, there is no core schema, either formal or informal.

observability records do not even need to have a timestamp, in the sense of a log message. for example, see relations imported from a relational database into observability records, in parsl monitoring import (crossref usecase about plotting from monitoring database)

Parsl contains two “mini-workflow-systems” on top of core Parsl: parsl-perf and pytest tests. It could be interesting to illustrate how those fit in without being a core part of Parsl observability.

Parsl monitoring visualisation and Parsl logging are both completely unaware of the application level structure of the mini-workflows run by parsl-perf and pytest, beyond what is expressed to the DFK as DAG fragments: there’s nothing to separate out parsl-perf iterations, or pytest tests.

In the context of pytest, see: *Adventure: pytest observing interchange variables*

Colmena

CHAPTER

THREE

GENERATING WIDE RECORDS

3.1 What exists now

3.1.1 Parsl

Parsl generates a lot of observability-style events, but spread across many different formats.

Parsl logs - not well structured, for example overlapping DFKs are not well represented, and actions to do with different tasks can be interleaved without being clearly separated/identified. Globus Compute deliberately makes them even less structured, by jumbling up the file-based logs of multiple runs into one directory

Parsl monitoring - well structured but very hard to modify. It is easy to query for questions that it can answer, and hard to use for anything more. Users are generally interested in using it when they discover it, but it suffers from a history of being built as student demo projects dropped into production. An example of a question it cannot answer: What is the `parsl_resource_specification` for a task?

Work Queue Logs - well structured. Ignored by Parsl monitoring. Hard to correlate with monitoring.db: Work Queue uses work queue task IDs, but Parsl Monitoring uses Parsl Task and Try IDs. Correlating those is a motivating use case for this Observabilty projected.

[TODO: make that correlation as an explicit use-case section, explaining what needs to change, how it is done manually now]

3.1.2 Academy

As a more in-development project, it is much better placed to make observability records from the start as a first-order production feature.

3.2 New Python Code for log generation

Acknowledging observability as a first-order feature means we can make big changes to code.

Every log message needs to be visited to add context. In many places a bunch of that context can be added by helpers: for example, in my prototype, some module level loggers are replaced by object-level loggers: there is a per-task logger (actually LoggerAdapter) in the TaskRecord, and logging to that automatically adds on relevant DFK and task metadata: at most log sites, the change to add that metadata is to switch from invoking methods on the module-level logger object, invoking them on the new task-level logger instead.

Some log lines bracket an operation, and to help with that, my prototype introduces a LexicalSpan context manager which can be used as part of a *with* block to identify the span of work starting and ending.

Move away from forming ad-hoc string templates and make log calls look more machine-readable. This is somewhat stylistic: with task ID automatically logged, there is no need to substitute in task ID in some arbitrary subset of task-related logs.

TODO: describe academy style that I tried out in PR #NNN:

```
extra=myobj.log_extra() | { "some": "more" }
```

Parsl config hook for arbitrary log initialization - actually it can do "anything" at process init and maybe that's interesting from a different perspective (because its a callback/plugin), but from the perspective of this report I don't care about non-log uses.

Be aware that there are non-Python bits of code generating various logs. Work Queue (still structured) logs are one example. The output from batch command submit scripts are another less structured one that looks much more like a traditional chaotic output file.

3.2.1 Python API on logging side

Use Python-provided logger interface, with Python-provided `` `extra` `` API.

per-class "log extras" method that generates an `extra` dict about this object. that pushes on (like `repr`) it being the responsibility of the object to describe itself, rather than being someone elses responsibility.

3.2.2 anonymous/temporary identified python objects

python objects don't have a global-over-time ID. id() exists but it is reused over time so awkward to use over a whole series of logs. so some objects should get a uuid "just" for observability - UUIDs were invented for this.

likewise e.g. gc endpoints don't have a DFK ID, but endpoint id/executor/block 0 isn't a global-over-time ID: there's a new block 0 at every restart? or is there a unique UEP ID each time that is enough? I don't think so because i see overlapping block-0 entries.

repr-of-ID-object might not be the correct format for logging: I want stuff that is nice strings for values, but repr (although it is a string) is more designed to look like a python code fragment rather than the core value of an object. Maybe `str` is better, and maybe some other way of representing the ID is better? The point is to have values that work well in aggregate, database style analysis, not easy on the human eye.

3.2.3 Contributed: Modifying academy to generate wide events

summarise the PRs I merged already

cross-ref event graph in analysis section as something enabled by this

3.3 Translating non-wide-event sources

Part of this modularity work is that some modules produce event-like information that looks superficially very different but that can be understood through the lens of structured event records.

3.3.1 Using Parsl monitoring events as wide logs

two approaches:

monitoring.json: abandons the SQL database component of conventional Parsl monitoring and instead writes each monitoring message out to a json file, giving an event stream.

replay-monitoring.db: turns a monitoring.db file into events. the status, resource and block tables already looks like an event stream. This gives an easy way to take existing runs and turn them into event streams without needing to opt-in to any of the other JSON logging, or changing anything at all at runtime: anything new is entirely post-facto. which fits the general concept of doing things post-facto in parsl observability.

the infrastructure for this already exists, which means that the query side of this project can be used without modification of the execution-side Parsl environment.

see earlier use case on priority visualization

3.3.2 Using Work Queue `transaction_log` as a wide log source

this is a core part of seeing beyond the pure-Parsl code. it's well structured but not JSON. translation into JSON is mostly syntactic. and can be done line-by-line, aka streaming.

TODO: example log line

note that work queue task IDs are not Parsl task IDs: data from the monitoring database cannot be correlated with data from the work queue transaction log! (without further help from the parsl JSON log files…)

3.4 Adventure: adding observability to a prototype: idris2interchange

Another example: swap out interchange impl for a different one with a different internal model: a schema of events for task progress through the original interchange doesn't necessarily work for some other implementation.

idris2interchange - i want to debug stuff, not be told by the observability system HAHA we don't support your prototyping. in some sense thats exactly the time I *need* the observability system to be helping me. not later on when it all works.

idris2interchange project is not aimed at producing production code. *ever*. in that sense it is very similar to some student projects that interact with parsl.

mini-journal: what did i have to do to support idris2 logging? * make log records JSON format instead of textual - prior format was timestamp / string. theres a json library but to start with this records are so simple i'll template them in. * also already had a simple log-of-value mechanism in there already which readily translates to logging a template, a full message, and the value as separate fields.

now there are json records going to the console. I don't trust the string escaping, but i'll deal with that ad-hoc. but also: needs to go to a file; if i want it to interact with other log files, I need some common keys. htex_task_id is the obvious one there for task correlation. manager ID is another.

To go to a file: lazy redirect of stdout to idris2interchange.log. This could be done more seriously to avoid random prints going to the file but this is a prototype so I don't care.

Run it through *jq* for basic validation and haha its broken. I got confused about JSON quotes vs Python style quotes. Various iterations of *jq* vs formatting fixes to work towards *jq* believing this is valid.

That log escaping, which i implemented pretty quickly, seems to make logging extremely slow - especially outputting the pickle stack which is actually quite a big representation when it has a manager registration with all my installed python packages in there. but hey thats what log levels/log optionality is for.

Let's do some scripting to figure out which of these lines is so expensive - based on line length. one line is 49kb long! (its repeating the full pickled task state rather than a task id!). and similar with manager IDs. but this is probably the sort of changes I'll be needing to make to tie stuff in with other log files anyway.

This log volume has been a problem for me elsewhere, even without structured logging, filling up eg my root filesystem with docker stdout logs.

Now back to `jq` validation…

if i get that done… look for every logv call and report each one and how many times it logged a value. this is in the direction of logging metrics, without actually being that.

a pytest run now give 92000 idris2interchange log lines.

and now jq accepts it all.

so lets see if parsl.obserability.load_jsons can load it. it can, without further change.

logs that have a v:

Next step is to figure out how task processing can be annotated to fit into the general task flow findcommon style output. Let's start with a single line such as this without trying to add any broader context.

Make a new logv that lets the v field be named. That allows a single association to be made. which is ok for this stage.

First lets format that task ID properly, without 'PickleInteger' in the value.

so now log records look like this:

which I hope is enough to align with the rest of findcommon.

So add in an import for this log into getlogs and try it out:

and there it is.

Next, I'd like to get more in here. Specifically of interest for observability development is I'd like to get an event for the point where a task message is received - even though at that point, its the beginning of a span that we won't know the task identity for until much later when the payload has been depickled and the task_id extracted.

The approach is probably something like a two parter: - make some span concept that has identities for all of its messages - tie that span to a task ID so that all its lines can get an htex_task_id widened on

This is an example of sending a join back in time. and an example of having to have the definition *somewhere* that these things are related - but that it doesn't have to be in the logging code where we prefer to be fast and stateless. Also a library call that finds an htex task id on any record of a group and widens out all the others to have the same id: look for "these keys" in groups identified by "these keys" and make them global. (`` `widen_implication` `` or some functional-dependency related name?). in this case, for the interchange log file, `` `submit_pass_id` `` => `` `htex_task_id` ``, or if doing so at a higher level `` `(dfk,executor,submit_pass_id)=>htex_task_id` ``

TODO: show task1 output before join. then implement join and show task1 output with the rest of the decode span in there - the deserialisation of the task and execution of the matchmaker is shown now.

TODO: add in result handling span in the same way.

widening submit_pass_id using key implication widening after loading/processing all the logs normally, which is what I'd expect if adding in ad-hoc hack stuff outside of the core parsl log loaders… has revealed some fixpoint related stuff: widening to htex_task_id which is the actual known ID isn't sufficient because the widening of htex_task_id to parsl_task_id already happened. I can widen to parsl_task_id OK because that implication has happened on the two log lines that already have an htex task ID. Is that ok in general? do I need fixpoints in general? something to keep an eye on. I think: as long as there is one record to convey the join as having happened, then a subsequent join can flesh that out. but if the join involves facts that aren't represented incrementally like that, then no. probably I can contrive some examples.

3.5 Performance measurement of patch stack on 2025-10-27

```
pip install -e . && parsl-perf --config parsl/tests/
↪configs/htex_local.py --iterate=1,1,1,10000
```

Running parsl-perf with constant block sizes (to avoid queue length speed changes):

master branch (165fdc5bf663ab7fd0d3ea7c2d8d177b02d731c5) 1139 tps

more-task-tied-logs: 1024

json-wide-log-records: 537

- but without initializing the JSONHandler: 1122

end of branch with all changes up to now: 385

3.6 Idea: Parsl resource monitoring on a host-wide basis

Ignore Parsl Monitoring per-task resource monitoring and do something else that generates similar observability records. This was always some disappointment with getting WQ resource monitoring into the Parsl monitoring database: what exists there that could be imported? Likewise, host-wide stuff doesn't fit well into the current Parsl Monitoring model but might fit better into an observability model.

3.7 Idea: worker node dmesg

especially for catching OOM Killer and other interesting kernel events that affect processes without giving user-expected stack traces.

Is dmesg available to users on Aurora worker nodes?

`mesg` already outputs JSON, if run with the right parameter.

That should be an hour to prototype alongside workers. Cross reference host-wide process monitoring as thematically related.

3.8 Idea: automatic instrumentation

Projects like OpenTelemetry offer automatic instrumentation. that would be interesting to experiment with here.

CHAPTER

FOUR

MOVING WIDE RECORDS AROUND

Events are generated in various places (for example, in application code running on HPC cluster worker nodes) and usually the user wants them somewhere else - for some kind of analysis, in the broad sense (*Analysing wide records*).

Concrete mechanisms for moving events around to a place of analysis should not be baked into the architecture. realtime options, file based options, … - that is an area for experimentation and this work should facilitate that rather than being prescriptive.

This chapter talks about different mechanisms that are on offer, and about how configuration might work. It tries to stay away from implementing much new mechanism, but rather tries to focus on integration of what already exists.

4.1 Comparison to Parsl logging

also followed by Academy at time of writing

expectation is you send your debugging expert a tarball of logs for them to pore over - this is extremely asynchronous but a very effective way of moving those records around. it has relatively low effect on performance behaviour: get the logs onto some filesystem while the performance-critical bit is running, move them from there later on.

- poring over these logs “later” - there’s no need for those logs to accumulate in real time in one place for post-facto analysis. And in practice, when doing log analysis rather than monitoring analysis, “send me a tarball of your runinfo” is a standard technique.

async movement is much easier than synchronous/realtime movement

4.2 Comparison to Parsl Monitoring

The transmission model is real-time. Even with recent radio plugins, the assumption is still that messages will arrive soon after being sent.

The almost-real-time data transmisison model is especially awkward when combined with SQL: distributed system events will arrive at different times or in the original UDP model perhaps not at all, and the “first” message that creates a task (for the purposes of the database) might arrive after some secondary data that requires that primary key to exist. yes, it’s nice for the SQL database to follow foreign key rules, especially when looking at the data “afterwards” but that’s not realistic for distributed unreliable events.

pluggable radios - inspiring following configurability model

Radios that exist currently: (TODO: upsides, downsides)

- UDP: sends UDP packets at submit side. UDP is unreliable. So events *do* get lost. I think the assumption at implementation time was that UDP packet loss is just some thing your professor tells you about, but clearly doesn’t happen.
- filesystem: needs a shared filesystem. One file = one monitoring event. If your filesystem is slow, which it often is, this is slow too.
- HTEX: the most efficient, but only works with the High Throughput Executor. Uses the existing HTEX result channel to send back monitoring events.

- ZMQ: This is over TCP. Like UDP radio, needs to be able to connect to the submit side. Probably better than UDP, although there's a TCP and ZMQ session initialization needed at the start of every task, because this radio does not persist connections across tasks. Unlike UDP, on the submit side, is yet another per-worker file descriptor use, I think, which is a serious scalability limitation.
- Python multiprocessing: for sending monitoring events within the same cluster of Python *multiprocessing* proceses: roughly the set of processes that were forked locally by the submit script using *multiprocessing*, so in htex: not the workers and not the interchange.

None of these are suitable for cloud-style environments where there is neither a shared filesystem or clean IP network. So I also prototyped an Academy radio for use with GC+Parsl - although I would rather use something like Octopus or Chronolog.

4.3 Python Configurability

A soft start in Parsl is to let people opt into observability style logs - with most performance hit coming from turning on json output, i think, it doesn't matter performance-wise too much about adding in the extra stuff on log calls.

The current parsl stuff is not set up for arbitrary log configuration outside of the submit-side process: for example, the worker helpers don't do any log config at all and rely on their enclosing per-executor environments to do it, which i think some do not.

htex interchange and worker logs have a hardcoded log config with a single debug boolean.

I'd like to do something a bit more flexible than adding more parameters, that reflect that in the future people might want to configure their handlers differently rather than using the JSONHandler.

eg. chronolog. pytest metrics observation in other section.

see Parsl monitoring radios configuration model. start prototyping that. note that it doesn't magically make arbitrary components that aren't compliant+Python redirectable. but thats fine in the modular approach.

see the existing initialize_logging which allows arbitrary user configurability at the submit-process side, by getting parsl completely out of the way and allowing the user to run whatever code they want.

4.4 Adventure: Wide records stored as JSON in files

This prototype stores Parsl logs that have been sent into the Python `logging` system as JSON objects, one per line.

This is one of the initial usecases for above configurability.

This was implemented is a straightforward Python logging *Handler* similar to the existing log handlers, the difference being how the output line is formatted.

The files are then moveable using traditional means: for exmaple, the classic "send me a tarball of your run directory".

4.5 Moving in realtime

What does realtime mean in this case? mostly a case of what do the ultimate consumers need, rather than any strong technical definition at this stage.

Inside Python parts of Parsl, this data *is* available in realtime at the point of logging as it goes to whatever LogHandler is running in each python process. that isn't true in general on the "event model" side of things, though.

Parsl already moves some event stuff around the network in realtime: that is the purpose of the monitoring radio system.

following two sections, octopus and chronolog, will talk about doing that.

My initial log-related work was post-facto: copy log files around. but there are plenty of mechanisms that should be able to deliver and analyse live, eg built around Diaspora Octopus

4.6 Adventure: Diaspora Octopus

This is an obvious follow-on to file-based JSON logs: the developers still kinda exist, and are friendly.

with Ryan and Haochen

This turned into a monster debugging and restructuring session around Octopus reliability.

Ryan has a specific use case he's trying to implement, that I am helping him with:

> i mostly want to know when my agents perform their loop so i can hackily use this as a heartbeat to determine if my agents alive, and, when the agent decides to call the llm, i want to know the outcome of that call
>
> —Ryan, on Slack

4.7 Idea: Chronolog

Nishchay did some stuff here. I don't know what the state is.

https://grc.iit.edu/research/projects/chronolog/

4.8 Adventure: pytest observing interchange variables

pytest htex task priority test wants to wait for interchange to have all the submitted tasks - which happens asynchronously to submit calls returning. it does that by logfile parsing. how does that fit into this observability story: there's a metric in my prototype for this value (which I used in one of the other use cases here).

Can do this by re-parsing the interchange log value. also could (with suitable configuration) attach a "pytest can see only metrics" log writer that runs over a unix socket? in some sense, injecting the relevant observability path into the interchange code as a configured log handler. that gives some motivation for the configurability section.

Also attaching a JSON log file to the interchange, and having a tail reader of that. also needs special configuration of interchange I think.

4.9 Adventure: Academy agents can report their own relevant logs via action

A prototype I made for Logan, and also showed to Ryan.

This ties in with Ryan's Diaspora use case for examining what individual agents are up to.

CHAPTER

FIVE

ANALYSING WIDE RECORDS

At small enough scale, which is actually quite a large number of tasks, given the volumes of data involved, parsing logs into a Python session and using standard Python tools like list comprehensions is a legitimate way to analyse things - rather than treating this approach as something awkwardly shameful that will be replaced by The Real Thing later. This is especially appealing to Parsl users who tend to be Python data science literate anyway.

That isn't the only approach and this is also modular: you get the records and can analyse them using whatever tools you personally find appropriate.

5.1 Adventure: All events for a task, in two aspects/presentations

TODO: move this from elsewhere, tidyup/modularise the code so it is presentable as a good first example. Show what it looks like: with monitoring.db import only; with full observability prototype.

emphasise: the first one is available now without needing to modify Parsl core.

emphasise: the same analytics code gives different results without needing much modification, given different *aspects* / *presentations* of the same run - see *Optional and missing data in observability*

emphasise: integration: there are resource records for tasks via monitoring, and htex internals via JSON logs: neither is a superset of the other.

5.1.1 Task events from the monitoring.db presentation

Import the monitoring.db created by unmodified master Parsl, and see how many event records there are:

```
>>> from parsl.observability.import_monitoring_db␣
↪import import_db
>>> l=import_db("runinfo/monitoring.db")
>>> len(l)
14596
```

Here's an example event record:

```
>>> l[0]
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 0, 'parsl_try_id': 0, 'parsl_
↪task_status': 'pending', 'created': 1764589563.
↪205463}
```

Now identify all the tasks, keyed hierarchically by DFK ID and task number:

```
>>> tasks = { (event['parsl_dfk'], event['parsl_
↪task_id']) for event in l if 'parsl_dfk' in event␣
↪and 'parsl_task_id' in event }
>>> len(tasks)
2432
```

and pick one randomly:

```
>>> import random
>>> random.choice(list(tasks))
('a08cc383-927a-4ce8-926b-f31e52e6edc2', 72)
```

That's task 72 of run `a08cc383-927a-4ce8-926b-f31e52e6edc2`.

Now pick out all the records that are labelled as part of that task, and print them nicely:

```
>>> events = [event for event in l if event.get(
↪'parsl_dfk', None) == 'a08cc383-927a-4ce8-926b-
↪f31e52e6edc2' and event.get('parsl_task_id',␣
↪None) == 72]


>>> for event in sorted(events, key=lambda event:␣
↪float(event['created'])): print(event)
...
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 72, 'parsl_try_id': 0, 'parsl_
↪task_status': 'pending', 'created': 1764589576.
↪775231}
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 72, 'parsl_try_id': 0, 'parsl_
↪task_status': 'launched', 'created': 1764589577.
↪435498}
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 72, 'parsl_try_id': 0, 'parsl_
↪task_status': 'running', 'created': 1764589577.
↪472009}
{'parsl_try_id': 0, 'parsl_task_id': 72, 'parsl_dfk
↪': 'a08cc383-927a-4ce8-926b-f31e52e6edc2',
↪'created': 1764589577.514019, 'resource_
↪monitoring_interval': 1.0, 'psutil_process_pid':␣
↪10680, 'psutil_process_memory_percent': 1.
↪2360561455881223, 'psutil_process_children_count
↪': 1.0, 'psutil_process_time_user': 1.26, 'psutil_
```

(continues on next page)

(continued from previous page)

```
↪process_time_system': 0.21000000000000002,
↪'psutil_process_memory_virtual': 416145408.0,
↪'psutil_process_memory_resident': 203653120.0,
↪'psutil_process_disk_read': 32585809.0, 'psutil_
↪process_disk_write': 20895.0, 'psutil_process_
↪status': 'sleeping', 'psutil_cpu_num': '3',
↪'psutil_process_num_ctx_switches_voluntary': 59.0,
↪ 'psutil_process_num_ctx_switches_involuntary':␣
↪396.0}
{'parsl_try_id': 0, 'parsl_task_id': 72, 'parsl_dfk
↪': 'a08cc383-927a-4ce8-926b-f31e52e6edc2',
↪'created': 1764589577.566039, 'resource_
↪monitoring_interval': 1.0, 'psutil_process_pid':␣
↪10680, 'psutil_process_memory_percent': 1.
↪2366030730861701, 'psutil_process_children_count
↪': 1.0, 'psutil_process_time_user': 1.28, 'psutil_
↪process_time_system': 0.22, 'psutil_process_
↪memory_virtual': 416145408.0, 'psutil_process_
↪memory_resident': 203661312.0, 'psutil_process_
↪disk_read': 32995622.0, 'psutil_process_disk_write
↪': 26441.0, 'psutil_process_status': 'sleeping',
↪'psutil_cpu_num': '3', 'psutil_process_num_ctx_
↪switches_voluntary': 69.0, 'psutil_process_num_
↪ctx_switches_involuntary': 454.0}
{'parsl_try_id': 0, 'parsl_task_id': 72, 'parsl_dfk
↪': 'a08cc383-927a-4ce8-926b-f31e52e6edc2',
↪'created': 1764589577.600372, 'resource_
↪monitoring_interval': 1.0, 'psutil_process_pid':␣
↪10680, 'psutil_process_memory_percent': 1.
↪2366030730861701, 'psutil_process_children_count
↪': 1.0, 'psutil_process_time_user': 1.3, 'psutil_
↪process_time_system': 0.23, 'psutil_process_
↪memory_virtual': 416145408.0, 'psutil_process_
↪memory_resident': 203444224.0, 'psutil_process_
↪disk_read': 33404930.0, 'psutil_process_disk_write
```

(continues on next page)

(continued from previous page)

```
↪': 33922.0, 'psutil_process_status': 'sleeping',
↪'psutil_cpu_num': '3', 'psutil_process_num_ctx_
↪switches_voluntary': 74.0, 'psutil_process_num_
↪ctx_switches_involuntary': 465.0}
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 72, 'parsl_try_id': 0, 'parsl_
↪task_status': 'running_ended', 'created':␣
↪1764589577.646802}
{'parsl_dfk': 'a08cc383-927a-4ce8-926b-f31e52e6edc2
↪', 'parsl_task_id': 72, 'parsl_try_id': 0, 'parsl_
↪task_status': 'exec_done', 'created': 1764589577.
↪671692}
```

So what comes out is some records which reflect the change in the task status as seen by the monitoring system (Which includes `running` and `running_ended` status that isn't known to the DFK) and while the task is running, some resource monitoring records.

This is basically a reformatting of records you could get by running SQL queries against `monitoring.db`, which is unsurprising: the only data source used was that database file.

5.1.2 Task events from a JSON logging presentation

I'm going to look at that same task (task 72 of run `a08cc383-927a-4ce8-926b-f31e52e6edc2`) from the perspective of JSON logs now – Parsl modified to output richer log files in JSON format with more machine readable metadata. TODO: reference the relevant generating events section.

All I am going to change is the importer command. I'm going to use the same selector and printing code shown above. So what we should get is a different *presentation* or *aspect* of the same task.

5.2 Adventure: The minimal change necessary to get htex task logs into the above task trace

This is probably one log line? To perform the join between IDs? Plus JSON logs.

TODO: present the changes I've done, but minimised to only this one change.

5.3 Adventure: blog: Visualization for task prioritisation

(two graphs that are already in parsl-visualize but probably-buggy - see #4021)

this uses replay-monitoring.db approach with no runtime changes. because the work I did there was in parsl master, but I want to do custom visualizations.

[TODO: link to blog post]

work towards a second blog post here. now most of the mechanics are worked out.

Step 2 of that: This was a second requirement on prioritisation from DESC.

use an A->B1/B2->C three step diamond-dag because its a bit less trivial.

visualization of task types for jim's follow on question: how can we adapt step 1 to colour by app name? It's not well presented in parsl-visualize because that focuses on state transitions rather than on app identity as the primary colour-key.

Visualisation also coloured by task-chain/task-cluster to show a cluster based visualization.

priority modes: natural (submit-to-htex order, “as unblocked” order), random (priority=random.random()), chain priority by chain depth, chain priority by cluster. the last two should be “the same” in plot 4 i hope. unclear what random mode will do, if anything? i guess get more later-unlocked tasks randomly in there? random is always interesting to me as pushing things away from degenerate cases - i this case “Cs run last”

plot 1: task run/running_ended individual tasks, coloured by parsl app name

plot 2: tasks of each of two kinds, coloured by parsl app name

plot 3: tasks running by type, with no priority, with two different priority schemes.

plot 4: Visualisation of end-result completed - i.e. how many C tasks have completed over time, ignoring everything else about the inside. with prioritisation and with my two prioritisation schemes.

Plot 4 should be the top level plot set - because it an example “goal” of the prioritisation, I think. (might be because you want results sooner, might be because C completing means you can delete a load of intermediate temporary data sooner).

From an observability perspective: the task chain identity is not known to Parsl. this is additional metadata, that in observability concepts, is added on by a “higher level system” and joined on at analysis time. the application knows about it, and the querier knows about it. none of the intermediate execution or observability infrastructure knows about it.

1. the status table rerun gives runtimes for plotting based on Parsl level dfk/task/try but doesn’t give any metadata about those. such as app name. in SQL this is added on as a JOIN, and so it is here too - rerun the tasks table as a sequence of log records - note that they don’t have a notion of “created” here because they are records but aren’t from a point in time, instead an already aggregated set of information. don’t let that scare you. observability records don’t have to look like the output of a printf!

5.4 Task flow logs through the whole system

Here's a use case that is hard with what exists in master-branch Parsl right now.

I want to know, for a particular arbitrary task, the timings of the task as it is submitted by the user workflow, flows through the DFK, into the htex interchange, worker pool, executes on an htex worker, and flows back to the user, with the timing of each step.

What exists in master Parsl right now is some information in monitoring, and some information in log files. The monitoring information is focused on the high level task model, not what is happening inside Parsl to run that high level model. Logs as they exist now are extremely ad-hoc, spread around in at least 4 different places, and poorly integrated: for example, log messages sometimes do not contain context about which task they refer to, do not represent that context uniformly (e.g. in a greppable way) and are ambiguous about context (e.g. some places refer to task 1, the DFK-level task 1, and some places refer to task 1, the HTEX-level task 1, which could be something completely different).

As a contrast, an example output of this prototype (as of 2025-10-26) is:

```
=== About task 358 ===
2025-10-26 10:29:46.467298 MainThread@117098 Task␣
↪358: will be sent to executor htex_Local (parsl.
↪log)
2025-10-26 10:29:46.467412 MainThread@117098 Task␣
↪358: Adding output dependencies (parsl.log)
2025-10-26 10:29:46.467484 MainThread@117098 Task␣
↪358: Added output dependencies (parsl.log)
2025-10-26 10:29:46.467550 MainThread@117098 Task␣
↪358: Gathering dependencies: start (parsl.log)
2025-10-26 10:29:46.467620 MainThread@117098 Task␣
```

(continues on next page)

(continued from previous page)

```
↪358: Gathering dependencies: end (parsl.log)
2025-10-26 10:29:46.467685 MainThread@117098 Task␣
↪358: submitted for App random_uuid, not waiting␣
↪on any dependency (parsl.log)
2025-10-26 10:29:46.467752 MainThread@117098 Task␣
↪358: has AppFuture: <AppFuture at 0x7f8bc1aed730␣
↪state=pending> (parsl.log)
2025-10-26 10:29:46.467818 MainThread@117098 Task␣
↪358: initializing state to pending (parsl.log)
2025-10-26 10:29:46.469992 Task-Launch_0@117098␣
↪Task 358: changing state from pending to launched␣
↪(parsl.log)
2025-10-26 10:29:46.470113 Task-Launch_0@117098␣
↪Task 358: try 0 launched on executor htex_Local␣
↪with executor id 340 (parsl.log)
2025-10-26 10:29:46.470240 Task-Launch_0@117098␣
↪Task 358: Standard out will not be redirected.␣
↪(parsl.log)
2025-10-26 10:29:46.470310 Task-Launch_0@117098␣
↪Task 358: Standard error will not be redirected.␣
↪(parsl.log)
2025-10-26 10:29:46.470336 MainThread@117129 HTEX␣
↪task 340: putting onto pending_task_queue␣
↪(interchange log)
2025-10-26 10:29:46.470404 MainThread@117129 HTEX␣
↪task 340: fetched task (interchange log)
2025-10-26 10:29:46.470815 Interchange-
↪Communicator@117144 Putting HTEX task 340 into␣
↪scheduler (Pool manager log)
2025-10-26 10:29:46.471166 MainThread@117162 HTEX␣
↪task 340: received executor task (Pool worker log)
2025-10-26 10:29:46.492449 MainThread@117162 HTEX␣
↪task 340: Completed task (Pool worker log)
2025-10-26 10:29:46.492742 MainThread@117162 HTEX␣
↪task 340: All processing finished for task (Pool␣
```

(continues on next page)

(continued from previous page)

```
↪worker log)
2025-10-26 10:29:46.493508 MainThread@117129 HTEX␣
↪task 340: Manager b'4f65802901c6': Removing task␣
↪from manager (interchange log)
2025-10-26 10:29:46.493948 HTEX-Result-Queue-
↪Thread@117098 Task 358: changing state from␣
↪launched to exec_done (parsl.log)
2025-10-26 10:29:46.494729 HTEX-Result-Queue-
↪Thread@117098 Task 358: Standard out will not be␣
↪redirected. (parsl.log)
2025-10-26 10:29:46.494905 HTEX-Result-Queue-
↪Thread@117098 Task 358: Standard error will not␣
↪be redirected. (parsl.log)
```

This integrates four log files and two task identifier systems into a single sequence of events.

5.5 Algebra of rearranging and querying wide events

These are some of the standard patterns I've found useful enough and straightforward to turn into library functions in the *parsl.observabilty* module.

look at relational algebra for phrasing and concepts

5.5.1 functorial

These operations are *functorial* in the sense that they operate on individual event records without regard for the context those records are in.

Widening

widen-by-constant: if we import a new log file but we know its broader context some other way, perhaps because it came from a known directory inside a parsl rundir (eg work queue's)

relabelling - to make names align from multiple sources, or to add distinction from multiple sources

5.5.2 non-functorial

These operations are *not functorial*. For example, widening-by-implication copies information between event records that are somehow grouped into a collection together, so cannot be implemented on a record-by-record basis.

post-facto relationship establishment using grouping and key implication

used where you might use a `JOIN` in SQL.

notion of identity and key-sequences: eg. parsl_dfk/parsl_task_id is a globally unique identifier for a parsl task across time and space, and so is parsl_dfk/executor_label/block_number or parsl_dfk/executor_label/manager_id/worker_number – although manager ID is also (in short form) globally unique. this is distinct from the hierarchical relations between entities - although hierarchical identity keys will often line up with execution hierarchy.

peter buneman XML keys stuff did nested sequences of keys for identifying xml fragments, c. year 2000

joins can send info back in time: if we have a span but don't know which parsl task it belongs to at the start, only later, we can use joins to bring that information from the future.

5.5.2.1 keys imply key operator

[l_keys] implies [r_key] over [collection]:

- if any log selected by l_keys contains an r_key, that r_key is unique (auto-check-that) and should be attached to every log record selected by l_keys. Use case: widening the task reception span in idris2interchange to be labelled with htex_task_id
- this is functional dependency: https://en.wikipedia.org/wiki/Functional_dependency

fixpoint notions that might need to be incorporated into the query model in Python code (so that a fixpoint can be converged to across non-local widening queries - see idris2interchange usecase notes)

Lots of different identifier spaces, loosely structured, not necessarily hierarchical: for example, an htex task is not necessarily "inside" a Parsl task, as htex can be used outside of a Parsl DFK (which is where the notion of Parsl task lives). An htex task often runs in a unix process but that process also runs many htex tasks, and an htex task also has extent outside of that worker process: there's no containment relationship either way.

5.6 Adventure: Browser UI

What might a browser UI look like for this?

compare parsl-visualize. compare scrolling through logs, but with some more interactivity (eg. click / choose "show me logs from same dfk/task_id")

But the parsl-visualize UI is so limited, it only has a handful of graphs to recreate. And some of them do not make sense to me so I would not recreate them.

I am not super excited about building UIs but it would probably be interesting to build something simple that can do a few queries and graphs to demonstrate log analysis in clickable form.

And then I could put in the analyses I have made (other graphs/reports) too, and also have it work with academy logs right away. and be ready to pull in other JSON log files as a more advanced implementation motivating JSON/wide events.

Use python and matplotlib, no web-specific stuff, to promote people who have done local scripting putting new plots into the UI, and promote using the graph code from the visualiser in own local scripting.

Make it able to address a whole collection of monitoring.db runs at once - not only one chosen workflow.

Use a parsl-aware list of quasi-hierarchical key names to drive narrow-down UI:

eg: pick dfk. pick: parsl task, parsl try, or executor, then executor task, or executor then block then task.

htex instance/block/manager/worker/htex_task

htex instance/block/manager/worker is an execution location - how an htex instance is identified is different between real Parsl and GC: in real parsl, its a dfk id/executor label.

are all graphs relevant for all key selections? or should eg. a block duration/count graph only appear in certain situations? eg if we've focused on one task-try, does that mean… no block status info and so no block graph? graphs could be enabled by: "if you see records like this, this graph is relevant". That would allow eg. enabling htex or WQ specific plots if we see (with more merged info) some htex or WQ specific data. If we only see academy or GC logs, should only report about them.

Recreate block vs task count graph from Matthews paper.

Aim for first iteration to work against current monitoring.db format so it can be tried out in a separate install against production runs, distinct from all other observability work. Exensibility there right from the start to allow that to extend for importing new data and plugging in plots and reports about new data.

two obvious non-monitoring.db extensions: what's happening with managers in blocks. whats happening with work queue. these are both executor specific, and don't fit the monitoring.db schema so well. so clear demos of what could be done better.

5.6.1 Idea: Streaming-fold web UI

what operators can be build with a streaming-fold? to give live updates as logs come in. (eg tail from a filesystem in the simplest case)

joins are the hard bit there, I think - but a fundep operator is at least constrained in its behaviour: cache keyed-but-unjoined blocks, if we see a key record, emit the whole block and forget it.

spend 1 day prototyping this.

counters, lists, a few graph types, drop downs/select fields

5.7 python side query model

log entries by reference (in lists and dicts). deliberately mutating the only copy of a log record, stored across multiple structs, is a feature not a bug. lets us select and modify and then return to the original structure. avoids copying the actual log records ever (except when explicit) and instead deal in object references.

use comprehension-style query notation to be reminiscent of other query languages.

5.8 academy visualization

message diagram - include here - for multi agent generator example I made for Logan.

5.9 HTEX vs WQ questions

Some questions don't make sense outside of the fixed worker count model. That is the htex tradition, but htex MPI mode has moved away from that with tasks able to request arbitrary amounts of some resource (mpi ranks) and WQ is heavily built around that too - DESC applications commonly use memory rather than core count as their resource constrained on a node. It is a feature, not a bug, that these features are different across executors.

5.10 Other record storage systems

SQLite - I experimented with this as part of my earlier DNPC work, using SQL as a query language over JSON records.

5.11 type checking event schemas

there is no observability schema overall, but individual components will have definitions which may be formal or informal. so it might be useful to be able to perform type checking on output records.

there can be type-checking/linting gradual type check of emitted events though: a couple of places:

- generate the extras by helper functions that force the record have certain fields (which should be referred to in the generating logs section)
- linting rules about x implies y, that can be regarded as hard type checking rules or soft rules depending on how you invoke such a tool

5.12 Writing out JSON events (or other formats) after performing query work

because maybe you've got somewhere better to process these results than the Python runtime.

or maybe you processed them somewhere that wasn't python and now want them in Python. JSON as the interface layer.

CHAPTER

SIX

ADVENTURE: ACADEMY VS GLOBUS COMPUTE

This isn't well documented in general and I feel like logging is especially neglected (although only part of the complexity here: code deployment is another part.) So I will work on bashing the two together with a hostile commentary.

6.1 getting started

Built `cloudlump` docker image for me to run locally to keep my deployments isolated from each other and from my home directory, to force me to be a bit more explicit about how to get things to work.

This is enough to start a GC endpoint in one container and then run `assert gcs.Executor().submit(abs, -7).result() == 7`

```
>>> import globus_compute_sdk as s
>>> e=s.Executor(endpoint_id='5c7202b5-d022-4534-
↪a399-21d4356129be')
>>> # authorization happens here
>>> assert e.submit(abs, -7).result() == 7
```

In this environment, the only task reference is in `~/.globus_compute/uep.`

`5c7202b5-d022-4534-a399-21d4356129be.bac77271-9a60-cad7-c4e9-0eb689ddf4d1/GlobusComputeEngine-HighThroughputExecutor/block-0/beb8bc8bb003/worker_0.log` using htex task numbers. This is not correlated with anything visible on the submit side – which is a UUID globus compute task ID:

```
>>> f=e.submit(abs, -8)
>>> f.task_id
'8b3da38f-f889-4ae1-8d31-68ff2f830876'
```

Suggestion: how to correlate GC task IDs with htex task IDs. Note that htex task IDs are not unique within the endpoint directory because multiple HTEXs (over time) log into the same directory.

block IDs are also not unique because of the above log directory conflation.

Parsl work with extra debug info is likely to give more task information here, but all stil correlated by htex task ID, which as mentioned above, is not even unique within an endpoint.

6.2 Launching an academy agent

There are no academy agents defined by default in an Academy install. If I manually define one in my submit container (in a disposable place, not shared), what happens when I try to launch it?

This works to a remote endpoint, from one cloudlump container to another, which surprises me a bit because the MyAgent code must be being conveyed by some bowels of GC serialization…

```
import asyncio
import globus_compute_sdk as gc
import academy.agent as aa
import academy.manager as am
import academy.exchange as ae
```

(continues on next page)

(continued from previous page)

```
import concurrent.futures as cf

print("importing myagent main")

class MyAgent(aa.Agent):
  @aa.action
  async def seven(self):
      print(f"something for stdout from MyAgent
↪{self!r}")
      import os
      return (7, os.getpid(), os.uname())

if __name__ == "__main__":

  async def main():
      # async with await am.Manager.from_exchange_
↪factory(factory=ae.HttpExchangeFactory(auth_
↪method='globus', url="https://exchange.academy-
↪agents.org"), executors = cf.
↪ProcessPoolExecutor()) as m:
      async with await am.Manager.from_exchange_
↪factory(factory=ae.HttpExchangeFactory(auth_
↪method='globus', url="https://exchange.academy-
↪agents.org"), executors = gc.Executor(endpoint_id=
↪'5c7202b5-d022-4534-a399-21d4356129be')) as m:
          print(f"with manager {m}")
          h = await m.launch(MyAgent())
          print(f"launched agent with handle {h}")
          s = await h.seven()
          print(f"agent seven result is {s}")
          assert s[0] == 7

  asyncio.run(main())
```

but if I put the agent in its own agentcode.py module, then I get a remote deserialization error:

```
...
File "/venv/lib/python3.13/site-packages/dill/_dill.
↪py", line 452, in load
   obj = StockUnpickler.load(self)
File "/venv/lib/python3.13/site-packages/dill/_dill.
↪py", line 442, in find_class
   return StockUnpickler.find_class(self, module,␣
↪name)
          ~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^
↪^^^
ModuleNotFoundError: No module named 'agentcode'
```

which is consistent: in the first example, dill is used to convey the definitions; in the second case, pickle thinks it can do an `import` and so never gets to the point of dill conveying the definitions.

6.3 Looking at GC-endpoint-side academy logs

For example, I want to see the agent action invocations. There are two paths here I might expect:

- some kind of endpoint/worker level as the academy docs suggest running as a worker initialization in the process worker pool. GC workers (and indeed, HTEX workers) don't have a configuration interface that supports that well, although in pure this observability project is working towards that – see the configurability section nearer the start. I might expect that as part of general observability of *the whole system* rather than hoping that the *other components* are themselves separately debuggable.

agent-level log routing: start something at agent start, shut it down at agent end. There are two existing approaches:

- I've prototyped making agents able to capture "their own" logs and report them via an agent action. I prototyped this with Logan, and mentioned it elsewhere in this report.

- Alok added an initialize logging feature to manager launching of agents to insert a log file capture. There is no facility there for conveying the log file anywhere else.

These different approaches are not contradictory: the Python logging mechanism can cope with multiple log handlers.

A goal: I want to look at agent activity - logs or visualization? - of stuff on the submit side and the remote side.

For example: I want to run my fibonacci agent test and see the agent logging its internal state as it changes, as well as seeing the client reporting what it sees.

I should be able to use multiple approaches to demonstrate how these events can be retrieved and give the same (or similar) output on the analysis side, and the differences in characteristics of the approach would be interesting to comment on - with the logging and analysis code the same for all event-movement approaches.

Academy manager has an option to initialize remote logging at start of agent execution – that's one of two remote logging hooks that exists already (the other is on user-side in agent startup). so lets see how they compare. the main questions: how much ahead-of-startup logging do I get from one but not the other? how much do I want my own free-coding configurability? e.g. for formatting or octopus-style?

eg. manager.launch(init_logging) wants to only run once, even though it has logfiles and log levels specified per-agent. that's a process vs "entity" inconsistency to think about. also doesn't have an extralog specifier. rather than wiring in yet another one, look at my more general log configuration.

This doesn't having configurable format, and it captures bits of logs from other stuff (such as the parsl worker_log entries). which is desirable sometimes, but perhaps more on an endpoint-wide basis.

It looks like it also doesn't *deinitialize* logging - because its a hack scoped around the process, rather than a principled log bracketing.

Now look at if I make the agent initialize its own logging- especially addressing the rough edges about: deinitialization, parameterisation.

CHAPTER SEVEN

THE REST

7.1 Debugging monitoring performance as part of developing this prototype

findcommon tool - finds common task sequence for templated logs and outputs their sequence, like this:

First run parsl-perf like this:

```
parsl-perf --config parsl/tests/configs/htex_local.
↪py

[...]

==== Iteration 3 ====
Will run 58179 tasks to target 120 seconds runtime
Submitting tasks / invoking apps
All 58179 tasks submitted ... waiting for completion
Submission took 103.880 seconds = 560.059 tasks/
↪second
Runtime: actual 137.225s vs target 120s
Tasks per second: 423.967
Tests complete - leaving DFK block
```

which executes a total around 60000 tasks.

First, note that this prototype benchmarks on my laptop significantly slower than the contemperaneous master branch, at .

That's perhaps unsurprising: this benchmark is incredibly log sensitive, as my previous posts have noted - TODO: link to blog post and to R-performance report) - around 900 tasks per second on a 120 second benchmark. And this prototype adds a lot of log output. Part of the path to productionisation would be understanding and constraining this.

From that output above, it is clear that the submission loop is taking a long time: 100 seconds. With about 35 seconds of execution happening afterwards. The Parsl core should be able to process task submissions much faster than 560 tasks per seconds. So what's taking up time there?

Run findcommon (a could-be-modular-but-isn't helper from this observability prototype) on the result:

```
0.0: Task %s: will be sent to executor htex_local
0.00023320618468031343: Task %s: Adding output␣
↪dependencies
0.0004515730863634116: Task %s: Added output␣
↪dependencies
0.000672943356177761: Task %s: Gathering␣
↪dependencies: start
0.0008952160973877195: Task %s: Gathering␣
↪dependencies: end
0.0011054732824941516: Task %s: submitted for App␣
↪app, not waiting on any dependency
0.001316777690507145: Task %s: has AppFuture: %s
0.0015680651123983979: Task %s: initializing state␣
↪to pending
23.684763520758917: HTEX task %s: putting onto␣
↪pending_task_queue
23.68483662049256: HTEX task %s: fetched task
23.684863335335613: Task %s: changing state from␣
↪pending to launched
```

(continues on next page)

(continued from previous page)

```
23.6850573607536: Task %s: try %s launched on␣
↪executor %s with executor id %s
23.685248910492184: Task %s: Standard out will not␣
↪be redirected.
23.685424046734745: Task %s: Standard error will␣
↪not be redirected.
23.686276226995773: Putting HTEX task %s into␣
↪scheduler
23.686777094898495: HTEX task %s: received executor␣
↪task
23.687025900194147: HTEX task %s: Completed task
23.687268549254735: HTEX task %s: All processing␣
↪finished for task
23.687837933843614: HTEX task %s: Manager %r:␣
↪Removing task from manager
23.688483699079185: Task %s: changing state from␣
↪launched to exec_done
```

In this stylised synthetic task trace, a task takes an average of 23 seconds to go from the first event (choosing executor) to the final mark as done. That's fairly consistent with the parsl-perf output - I would expect the average here to be around half the time of parsl-perf's submission time to completion time (30 seconds).

What's useful with findcommon's output is that it shows the insides of Parsl's working in more depth: 20 states instead of parsl-perf's start, submitted, end. And the potential exists to calculate other statistics on these events.

So in this average case, there's something slow happening between setting the task to pending, and then the task "simultaneously" being marked as launched on the submit side and the interchange receiving it and placing it in the pending task queue.

That's a bit surprising - tasks are meant to accumulate in the interchange, not before the interchange.

So let's perform some deeper investigations – observability is for Serious Investigators and so it is fine to be hacking on the Parsl source code to understand this more. (by hacking, I mean making temporary changes for the investigation that likely will be thrown away rather than integrated into master).

Let’s flesh out the whole submission process with some more log lines. On the DFK side, that’s pretty straightforward: the observability prototype has a per-task logger which, if you have the task record, will attach log messages to the task.

For example, here’s the changes to add a log around the first call to launch_if_ready, which is probably the call that is launching the task.

```
+   task_logger.debug("TMP: dependencies added,␣
↪calling launch_if_ready")
    self.launch_if_ready(task_record)
+   task_logger.debug("TMP: launch_if_ready returned
↪")
```

My suspicion is that this is around the htex submission queues, with a secondary submission around the launch executor, so to start with I’m going to add more logging around that.

Then rerun parsl-perf and findcommon, without modifying either, and it turns out to be that secondary submission, the launch executor:

```
0.0020453477688227: Task %s: TMP: submitted into␣
↪launch pool executor
0.0022568703064342224: Task %s: TMP: launch_if_ready␣
↪returned
14.073021359217009: Task %s: TMP: before submitter␣
↪lock
[...]
14.078550367412324: Task %s: changing state from␣
↪launched to exec_done
```

Don’t worry too much about the final time (14s) changing from 23s in the earlier run – that’s a characteristic of parsl-perf batch sizes that I’m working on in another branch.

If that's the case, I'd expect the thread pool executor, previously much faster than htex, to show similar characteristics:

surprisingly, though although the throughput is not much much higher… the trace looks very different timewise. the bulk of the time here still happens at the same place, there isn't so much waiting there - less than a second on average. That's possibly because the executor can get through tasks much faster so the queue doesn't build up so much?

```
==== Iteration 2 ====
Will run 68976 tasks to target 120 seconds runtime
Submitting tasks / invoking apps
All 68976 tasks submitted ... waiting for completion
Submission took 117.915 seconds = 584.965 tasks/
↪second
Runtime: actual 118.417s vs target 120s
Tasks per second: 582.485
```

```
0.0: Task %s: will be sent to executor threads
0.00014157412110423425: Task %s: Adding output␣
↪dependencies
0.00028986527250472O1: Task %s: Added output␣
↪dependencies
0.000425118042214259: Task %s: Gathering␣
↪dependencies: start
0.0005696294991521399: Task %s: Gathering␣
↪dependencies: end
0.0006999648174108608: Task %s: submitted for App␣
↪app, not waiting on any dependency
0.0008433702196425292: Task %s: has AppFuture: %s
0.0010710284919573986: Task %s: initializing state␣
↪to pending
0.0011652027385929428: Task %s: TMP: dependencies␣
↪added, calling launch_if_ready
0.0012973675719411494: Task %s: submitting into␣
↪launch pool executor
```

(continues on next page)

(continued from previous page)

```
0.0014397921284467212: Task %s: submitted into␣
↪launch pool executor
0.0015767665501452072: Task %s: TMP: launch_if_
↪ready returned
0.3143575128217656: Task %s: before submitter lock
0.31448896150771743: Task %s: after submitter lock,␣
↪before executor.submit
0.3146383380777917: Task %s: after before executor.
↪submit
0.3147926810507091: Task %s: changing state from␣
↪pending to launched
0.3149239369413048: Task %s: try 0 launched on␣
↪executor threads
0.31504996538376506: Task %s: Standard out will not␣
↪be redirected.
0.31504996538376506: Task %s: Standard out will not␣
↪be redirected.
0.3151759985402679: Task %s: Standard error will␣
↪not be redirected.
0.3151759985402679: Task %s: Standard error will␣
↪not be redirected.
0.315319734920821: Task %s: changing state from␣
↪launched to exec_done
```

So maybe I can do some graphing of events to give more insight than these averages are showing. A favourite of mine from previous monitoring work is how many tasks are in each state at each moment in time. I'll have to implement that for this observability prototype, because it's not done already, but once it's done it should be reusable. and it should share most infrastructure with *findcommon*. Especially relevant is discovering where bottlenecks are: it looks like this is a parsl-affecting performance regression that might be keeping workers idle. For example, we could ask: does the interchange have "enough" tasks at all times to keep dispatching. With 8 cores on my laptop, I'd like it to have at least 8 tasks or so inside htex at any one time, but this looks like it might not be true. Hopefully graphing will reveal more. It's also important to note that this findcommon output shows latency, not throughput – though high latency at particular points is an indication of throughput problems.

Or, I can look at how many tasks are in the interchange over time: there either is, or straightforwardly can be, a log line for that. That will fit a different model to the above log lines which are per-task. Instead they're a metric on the state of one thing only: the interchange. of which there is only one, at least for the purposes of this investigation.

Add a new log line like this into the interchange at a suitable point (after task queueing, for example):

```
+  ql = len(self.pending_task_queue)
+  logger.info(f"TMP: there are {ql} tasks in the␣
↪pending task queue", extra={"metric": "pending_
↪task_queue_length", "queued_tasks": ql})
```

Now can either look through the logs by hand to manually see the value. Or extract it programmatically and plot it with matplotlib, in an ad-hoc script:

```
import matplotlib.pyplot as plt
from parsl.observability.getlogs import getlogs

logs = getlogs()

# looking for these logs:
# "metric": "pending_task_queue_length", "queued_
↪tasks": ql})

metrics = [(float(l['created']), int(l['queued_tasks
↪']))
           for l in logs
           if  'metric' in l
           and l['metric'] == "pending_task_queue_
↪length"
          ]
```

(continues on next page)

(continued from previous page)

```
plt.scatter(x=[m[0] for m in metrics],
            y=[m[1] for m in metrics])

plt.show()
```

and indeed that shows that the interchange queue length almost never goes above length 1, and never above length 10.

That's enough for now, but it's a usecase that shows partially understanding throughput: we can see from this observability data that the conceptual 50000 task queue that begins in parsl-perf as a *for*-loop doesn't progress fast enough to the interchange internal queue, and so probably performance effort should probably be focused on understanding and improving the code path around launch and getting into the interchange queue. With an almost empty interchange queue, anything happening on the worker side is probably not too relevant, at least for that parsl-perf use case.

This "understand the queue lengths (or implicit queue lengths) towards execution" investigation style has been useful in understanding Parsl performance limitations in the past.

7.2 See also

NetLogger - https://dst.lbl.gov/publications/NetLogger.tech-report.pdf

my dnpc work, an earlier iteration of this. more focused on human log parsing and so very fragile in the face of improving log messages, and not enough context in the human component.

syslog, systemd logging, linux kernel ringbuffer/dmesg

buneman xml keys (mentioned above, c.2000) microsoft power bi: As a simple example of how do we get this data into something actually novel for academia. Dashboard friendly.

7.3 wheres the bottleneck - visualization

based on template analysis - but could be based on anything that can be grouped and identified.

7.4 Review of changes made so far to Parsl and Academy

This should be part of understanding what sort of code changes I am proposing.

7.5 Applying this approach for academy

As an extreme “data might not be there” – perhaps Parsl isn’t there at all. What does this code and these techniques look like applied to a similar but very different codebase, Academy, which doesn’t have any distributed monitoring at all at the moment. There are ~100 log lines in the academy codebase right now. How much can this be converted in a few hours, and then analysed in similar ways?

The point here being both considering this as a real logging direction for academy, and as a proof-of-generality beyond Parsl.

thoughts:

academy logging so far focused on looking pretty on the console: eg ANSI colour - that’s at the opposite end of the spectrum to what this observability project is trying to log.

rule of thumb for initial conversion: whatever is substituted into the human message should be added as an extras field.

CHAPTER EIGHT

ACKNOWLEDGEMENTS

chronolog: nishchay, inna

desc: esp david adams, tom glanzman, jim chiang

uiuc: ved

gc: kevin

academy: alok, greg, logan

diaspora: Ryan, Haochen

parsl: matthew

INDEX