FAQs

Some frequently asked questions about flux and their answers! 🤔

Background

What is Flux Framework?

Flux is a flexible framework for resource management, built for your site. The core framework here consists of a suite of projects, tools, and libraries that may be used to build site-custom resource managers for High Performance Computing centers. The set of core projects are described in this documentation, and our larger family of projects can be seen on our portal page.

What does it mean for a cluster to deploy Flux?

Most of the time when someone talks about Flux, they will be describing the combined install of several projects here that manifest in a full cluster to submit workflows. This cluster is comparable to other job managers like Slurm or SGE in that it can be installed as the main workload manager for a site.

Where does Flux work?

You likely are associating Flux with high performance computing in that it is comparable to other job managers. However, Flux has a unique ability to nest, meaning you (as a user) could launch a Flux Instance under a Slurm allocation, for example. Along with scheduler nesting, you can easily demo Flux in a container, or even used in Kubernetes with the Flux Operator. We have a vision for Flux to allow for converged computing, or making it easy to move between traditional HPC and cloud.

How does Flux compare to other managers?

Flux has similarities to many other resource managers, but we do a variety of things quite differently. Check out this comparison table here (Flux Compared to Other Re latest
a detailed list!" Do you want to add or correct a feature? Please let us know

General Questions

What’s with the fancy ƒ?

Flux job IDs and their multiple encodings are described in RFC 19. The `ƒ` prefix denotes the start of the F58 job ID encoding. Flux tries to determine if the current locale supports UTF-8 multi-byte characters before using `ƒ`, and if it cannot, substitutes the alternate ASCII `f` character. If necessary, you may coerce the latter by setting `FLUX_F58_FORCE_ASCII=1` in your environment.

Most flux tools accept a job ID in any valid encoding. You can convert from F58 to another using the flux-job(1) `id` subcommand, e.g.

```
$ flux submit sleep 3600 | flux job id --to=words
airline-alibi-index--tuna-maximum-adam
$ flux job cancel airline-alibi-index--tuna-maximum-adam
```

With copy-and-paste, auto-completion, globbing, etc., it shouldn’t be necessary to *type* a job ID with the `ƒ` prefix that often, but should you need to, use your terminal’s method for inputting a Unicode U+0192:

gnome terminal

Press `ctrl` + `shift` + `U` then type `0192` and press `space` or `enter`.

mac

Press `option` + `f`.

If your Konsole terminal displays `ƒ` as `Æ`, check that Settings → Edit → Profile → Advanced → Encoding: Default Character Encoding is set to `UTF-8`, not `ISO8859-1`.

Does flux run on a Mac?

Not yet. We have an open issue on GitHub tracking the progress towards the goal of natively compiling on a mac. In the meantime, you can use Docker, see: Quick Start.

Can you recommend a shell prompt?

It can sometimes be tricky dealing with Flux hierarchies of instances. A comı adjustment used by Flux users adds Flux resource size and instance depth ir latest example, the following prompt shows that we have 64 nodes of resources and are at depth 1.

```
[s=64,d=1] $
```

To add this prompt into your shell, you can cut and paste the below or use it to adjust your current shell prompt. Note that the initial call to `flux getattr size` is simply used to test if you are currently running on a system with Flux or not.

Cut and paste for `.bashrc`

```
flux getattr size > /dev/null 2>&1
if [ $? -eq 0 ]; then
    export PS1="[s=$(flux getattr size),d=$(flux getattr instance-level)] $"
fi
```

Cut and paste for `.cshrc`

```
flux getattr size >& /dev/null
if ( $? == 0 ) then
    set prompt="[s=`flux getattr size`,d=`flux getattr instance-level`] $"
endif
```

How do I report a bug?

You can read up on reporting bugs here: Contributing or report one directly for flux core or sched.

Resources Questions

Why is Flux ignoring my Nvidia GPUs?

When Flux is launched via a foreign resource manager like Slurm or LSF, it must discover available resources from scratch using hwloc. To print a resource summary, run:

```
$ flux resource info
16 Nodes, 96 Cores, 16 GPUs
```

The version of hwloc that Flux is using at runtime must have been configured with `--enable-cuda` for it to be able to detect Nvidia GPUs. You can test to see if hwloc is able to detect installed GPUs with:

```
$ lstopo | grep CoProc
```

latest

If no output is produced, then hwloc does not see any Nvidia GPUs.

This problem manifests itself differently on a Flux system instance where *R* (the resource set) is configured, or when Flux receives *R* as an allocation from the enclosing Flux instance. In these cases Flux checks *R* against resources reported by hwloc, and drains any nodes that have missing resources.

Why are resources missing in foreign-launched Flux?

When Flux discovers resources via hwloc, it honors the current core and GPU bindings, so if resources are missing, affinity and binding from the parent resource manager should be checked. In Slurm, try `--mpibind=off`, in LSF jsrun, try `--bind=none`.

How can I oversubscribe tasks to resources in Flux?

There are several ways to decouple a job's task count from the quantity of allocated resources, depending on what you want to do.

If you simply want to oversubscribe tasks to resources, you can use the per-resource options of the job submission commands instead of the more common per-task options. For example, to launch 100 tasks per node across 2 nodes:

```
$ flux run --tasks-per-node=100 -N2 COMMAND
```

The per-resource options were added in flux-core 0.43.0. In earlier versions, the same effect can be achieved by setting the `per-resource.` job shell options directly:

```
$ flux run -o per-resource.type=node -o per-resource.count=100 -N2 COMMAND
```

Another method to more generally oversubscribe resources is to launch multiple Flux brokers per node. This can be done locally for testing, e.g.

```
$ flux start -s 4
```

or can be done by launching a job with multiple `flux start` commands per node, e.g. to run 8 brokers across 2 nodes

```
$ flux submit -o cpu-affinity=off -N2 -n8 flux start SCRIPT
```

One final method is to use the `alloc-bypass` jobtap plugin, which allows a j latest scheduler entirely by supplying its own resource set. When this plugin is load can submit a job with the `system.alloc-bypass.R` attribute set to a valid Resource Set

Specification. The job will then be executed immediately on the specified resources. This is useful for co-locating a job with another job, e.g. to run debugger or other services.

```
$ flux jobtap load alloc-bypass.so
$ flux submit -N4 sleep 60
ƒ2WU24J4NT
$ flux run --setattr=system.alloc-bypass.R="$(flux job info ƒ2WU24J4NT R)" -n 4
flux getattr rank
3
2
1
0
```

How do I prevent Flux from filling up /tmp?

Flux's key value store is backed by an SQLite database file, located by default in *rundir*, typically `/tmp`. On some systems, `/tmp` is a RAM-backed file system with limited space, and in some situations such as long running, high throughput workflows, Flux may use a lot of it.

Flux may be launched with the database file redirected to another location by setting the *statedir* broker attribute. For example:

```
$ mkdir -p /home/myuser/jobstate
$ rm -f /home/myuser/jobstate/content.sqlite
$ flux batch --broker-opts=-Sstatedir=/home/myuser/jobdir -N16 ...
```

Or if launching via flux-start(1) use:

```
$ flux start -o,-Sstatedir=/home/myuser/jobdir
```

Note the following:

- The database is only accessed by rank 0 so *statedir* need not be shared with the other ranks.
- *statedir* must exist before starting Flux.
- If *statedir* contains `content.sqlite` it will be reused. Unless you are intentionally restarting on the same nodes, remove it before starting Flux.
- Unlike *rundir*, *statedir* and the `content.sqlite` file within it are not cleaned up when Flux exits.

See also: flux-broker-attributes(7).

latest

Jobs Questions

How do I efficiently launch a large number of jobs?

If you have more than 10K fast-cycling jobs to run, here are some tips that may help improve efficiency and throughput:

- Create a batch job or allocation to contain the jobs in a Flux subinstance. This improves performance over submitting them directly to the Flux system instance and reduces the impact of your jobs on system resources and other users. See also: Batch Jobs.
- If scripting `flux submit` commands, avoid the pattern of one command per job as each command invocation has a startup cost. Instead try to combine similar job submissions with `flux submit --cc=IDSET` or flux-bulksubmit(1).
- By default `flux submit --cc=IDSET` and `flux bulksubmit` will exit once all jobs have been submitted. To wait for all jobs to complete before proceeding, use the `--wait` or `--watch` options to these tools.
- If multiple commands must be used to submit jobs before waiting for them, consider using `--flags=waitable` and `flux job wait --all` to wait for jobs to complete and capture any errors.
- If the jobs to be submitted cannot be combined with the command line tools, develop a workflow management script using the Flux python interface. The flux-run command itself is a python program that can be a useful reference.
- If jobs produce a significant amount of standard I/O, use the flux-submit(1) `--output` option to redirect it to files. By default, standard I/O is captured in the Flux key value store, which holds other job metadata and may become a bottleneck if jobs generate a large amount of output.
- When handling many fast-cycling jobs, the rank 0 Flux broker may require significant memory and cpu. Consider excluding that node from scheduling with `flux resource drain 0`.

Since Flux can be launched as a parallel job within foreign resource managers like Slurm and LSF, your efforts to develop an efficient batch or workflow management script that runs within a Flux instance can be portable to those systems.

How do I run job steps?

A Flux batch job or allocation started with `flux batch` or `flux alloc` is actually a full featured Flux instance run as a job within the enclosing Flux instance. Unlike Slurm, Fl
separate concept like *steps* for work run in a Flux subinstance—we just have
batch script in Flux may contain multiple `flux run` commands just as a Slurm batch script may contain multiple `srun` commands.

Despite there being only one type of *job* in Flux, running a series of jobs within a Flux subinstance confers several advantages over running them directly in the Flux system instance:

- System prolog and epilog scripts typically run before and after each job in the system instance, but are skipped between jobs within a subinstance.
- The Flux system instance services all users and active jobs running at that level, but the subinstance operates independently and is yours alone.
- Flux accounting may enforce a maximum job count at the system instance level, but the subinstance counts as a single job no matter how many jobs are run within it.
- The user has full administrative control over the Flux subinstance, whereas "guests" have limited access to the system instance.

Flux's nesting design makes it possible to be quite sophisticated in how jobs running in a Flux subinstance are scheduled and managed, since all Flux tools and APIs work the same in any Flux instance.

See also: Batch Jobs.

Why is my job not running?

If flux-jobs(1) shows your job in one of the pending states, you can probe deeper to understand what is going on. First, run `flux-jobs` with a custom output format that shows more detail about pending states, for example:

```
$ flux jobs --format="{id.f58:>12} {name:<10.10} {urgency:<3} {priority:<12}
{state:<8.8} {dependencies}"
       JOBID NAME       URG PRI          STATE    DEPENDENCIES
   ƒABLQgbbf3d sleep      16  16           SCHED
   ƒABLQty9fSX sleep      16  16           SCHED
   ƒABLR7sqQkf sleep      16  16           SCHED
   ƒABLRJnt85u sleep      16  16           SCHED
   ƒABLRVunjfu sleep      16  16           SCHED
   ƒABLRgR7eVd sleep      16  16           SCHED
   ƒABLQJnzDfV sleep      16  16           RUN
```

The job state machine is defined in RFC 21. Normally a job advances from NEW to DEPEND, PRIORITY, SCHED, RUN, CLEANUP, and finally INACTIVE. A job can be blocked in any of the following states:

DEPEND

latest

The job is awaiting resolution of a dependency. A job submitted without explicit dependencies may still acquire them. For example, flux-accounting may add a `max-running-jobs-user-limit` dependency when a user has too many jobs running, and resolve it once some jobs complete.

PRIORITY

The job is awaiting priority assignment. Flux-accounting may hold a job in this state if the user's bank is not yet configured.

SCHED

The job is waiting for the scheduler to allocate resources. A job may be held this state indefinitely by setting its *urgency* to zero. Otherwise, the scheduler decides which job to run next depending on the job's *priority* value, availability of the requested resources, and the scheduler's algorithm.

Note that the job's priority value defaults to the urgency, but a Flux system instance may be configured to use the flux-accounting multi-factor priority plugin, which sets priority based on factors that include historical and administrative information such as bank assignments and allocations.

The job state transitions are driven by job *events*, also defined in RFC 21. Sometimes it is helpful to see the detailed events when diagnosing a stuck job. A job eventlog can be printed using the following command:

```
$ flux job eventlog --time-format=offset ƒABFhJBw1dh
0.000000 submit userid=5588 urgency=16 flags=0 version=1
0.014319 validate
0.027185 depend
0.027262 priority priority=16
```

This job is blocked in the SCHED state, having not yet received an allocation from the scheduler. Job events may also be viewed in real time when a job is submitted with `flux run`, for example:

```
$ flux run -vv -N2 sleep 60
jobid: ƒABKQfqHf3u
0.000s: job.submit {"userid":5588,"urgency":16,"flags":0,"version":1}
0.015s: job.validate
0.028s: job.depend
0.028s: job.priority {"priority":16}
0.036s: job.alloc {"annotations":{"sched":{"queue":"debug"}}}
0.037s: job.prolog-start {"description":"job-manager.prolog"}
```

latest

```
0.524s: job.prolog-finish {"description":"job-manager.prolog","status":0}
0.538s: job.start
```

Why is my running job stuck?

If a job is getting to RUN state but still isn’t getting started, it may be helpful to look at job’s exec eventlog, which is separate from the primary eventlog described in Why is my job not running?

```
$ flux job eventlog --path=guest.exec.eventlog --time-format=offset ƒABaWMZ7UmD
0.000000 init
0.004929 starting
0.348570 shell.init leader-rank=6 size=2 service="5588-shell-68203540434124800"
0.358706 shell.start task-count=2
2.360860 shell.task-exit localid=0 rank=0 state="Exited" pid=10034 wait_status=0
signaled=0 exitcode=0
2.416990 complete status=0
2.417061 done
```

These events may also be viewed in real time, combined with the primary eventlog when a job is submitted by `flux run`:

```
$ flux run -vvv -N2 sleep 2
jobid: ƒABaWMZ7UmD
0.000s: job.submit {"userid":5588,"urgency":16,"flags":0,"version":1}
0.015s: job.validate
0.028s: job.depend
0.028s: job.priority {"priority":16}
0.038s: job.alloc {"annotations":{"sched":{"queue":"debug"}}}
0.038s: job.prolog-start {"description":"job-manager.prolog"}
0.520s: job.prolog-finish {"description":"job-manager.prolog","status":0}
0.532s: job.start
0.522s: exec.init
0.527s: exec.starting
0.871s: exec.shell.init {"leader-rank":6,"size":2,"service":"5588-shell-
68203540434124800"}
0.881s: exec.shell.start {"task-count":2}
2.883s: exec.shell.task-exit
{"localid":0,"rank":0,"state":"Exited","pid":10034,"wait_status":0,"signaled":0,"
exitcode":0}
2.939s: exec.complete {"status":0}
2.939s: exec.done
2.939s: job.finish {"status":0}
```

Why does the `flux bulksubmit` command hang?

latest

The `flux bulksubmit` command works similar to GNU parallel or `xargs` and is likely blocked waiting for input from `stdin`. Typical usage is to send output of some command to `bulksubmit` and, like `xargs -I`, substitute the input with `{}`. For example:

```
$ seq 1 4 | flux bulksubmit --watch echo {}
ƒ2jBnW4zK
ƒ2jBoz4Gf
ƒ2jBoz4Gg
ƒ2jBoz4Gh
1
2
3
4
```

As an alternative to reading from `stdin`, the `bulksubmit` utility can also take inputs on the command line separated by `:::`.

The `--dry-run` option to `flux bulksubmit` may be useful to see what would be submitted to Flux without actually running any jobs

```
$ flux bulksubmit --dry-run echo {} ::: 1 2 3
bulksubmit: submit echo 1
bulksubmit: submit echo 2
bulksubmit: submit echo 3
```

For more help and examples, see flux-bulksubmit(1).

How do I run a batch or alloc job that is resilient to node failures?

TL;DR: Use:

```
$ flux batch --conf=tbon.topo=kary:0 -o exit-timeout=none ...
```

Note

In future versions of Flux `-o exit-timeout=none` may become the default for flux-batch(1) and flux-alloc(1). Check with your version to see if `-o exit-timeout=none` is necessary.

When a Flux instance running as a job loses a node, what happens next is dependent on two factors: whether the lost node is critical, and the value of the `exit-timeout` job shell option. If the lost node is critical, the instance can no longer properly function, triggering a fatal `node-failure` job exception. If the node is not critical, a non-fatal job exception is raised an latest is notified. After the `exit-timeout` period (if set to a value other than `none`), a fatal job exception is raised and the job is terminated.

To maximize resilience in batch or allocation jobs, disable the `exit-timeout` option (set it to `none`) and minimize the number of critical ranks by running a flat TBON with `--conf=tbon.topo=kary:0`. This configuration allows jobs to continue running even if any node is lost, except for rank 0, which is always critical.

The same rules apply to jobs running within the instance. For parallel jobs that are not instances of Flux, all ranks are considered critical by default. As a result, jobs running on the lost nodes within the instance are immediately terminated under normal circumstances. Additionally, the resource set available for scheduling new jobs is reduced by the lost nodes. This means that pending and newly submitted jobs requesting more resources than are currently available will trigger a fatal "unsatisfiable" job exception.

MPI Questions

How do I set MPI-specific options?

The environment that Flux presents to MPI is via the flux-shell(1), which is the parent process of all MPI processes. There is typically one flux shell per node launched for each job. A Flux shell plugin offers a PMI server that MPI uses to bootstrap itself within the application's call to `MPI_Init()`. Several shell options affect the PMI implementations.

verbose=2

> If the shell verbosity level is set to 2 or greater, a trace of the PMI server operations is emitted to stderr, which can help debug an MPI application that is failing within `MPI_Init()`.

pmi=off

> Disable all PMI implementations.

pmi=LIST

> By default, only `simple` PMI is offered. If shell plugins for additonal implementations are installed, like for `pmix` or `cray-pals`, set *LIST* to a comma-separated list of implementations to enable.

pmi-simple.nomap

> Populate neither the `flux.taskmap` nor `PMI_process_mapping` keys in t latest

In addition to the PMI server, the shell implements "MPI personalities" as lua scripts that are sourced by the shell. Scripts for generic installs of openmpi, Intel MPI are loaded by default from

`/etc/flux/shell/lua.d`. Other personalities are optionally loaded from `/etc/flux/shell/lua.d/mpi`:

mpi=none

> Disable the personality scripts that are normally loaded by default.

mpi=spectrum

> IBM Spectrum MPI is an OpenMPI derivative. See also Launching Spectrum MPI within Flux.

MPI personality options may be added by site administrators, or by other packages.

Example: launch a Spectrum MPI job with PMI tracing enabled:

```
$ flux run -ompi=spectrum -overbose=2 -n4 ./hello
```

What versions of OpenMPI work with Flux?

Flux plugins were added to OpenMPI 3.0.0. Generally, these plugins enable OpenMPI major versions 3 and 4 to work with Flux. OpenMPI must be configured with the Flux plugins enabled. Your installed version may be checked with:

```
$ ompi_info|grep flux
              MCA pmix: flux (MCA v2.1.0, API v2.0.0, Component v4.0.3)
            MCA schizo: flux (MCA v2.1.0, API v1.0.0, Component v4.0.3)
```

Unfortunately, an OpenMPI bug broke the Flux plugins in OpenMPI versions 3.0.0-3.0.4, 3.1.0-3.1.4, and 4.0.0-4.0.1. The fix was backported such that the 3.0.5+, 3.1.5+, and 4.0.2+ series do not experience this issue.

A slightly different OpenMPI bug caused segfaults of MPI in `MPI_Finalize` when UCX PML was used. The fix was backported to 4.0.6 and 4.1.1. If you are using UCX PML in OpenMPI, we recommend using 4.0.6+ or 4.1.1+.

A special job shell plugin, offered as a separate package, is required to bootstrap the openmpi 5.0.x releases. Once installed, the plugin is activated by submitting a job with the `-opmi=pmix` option. In fact, `-o pmi=pmix` also works for earlier versions of OpenMPI, and appears to avoid a UCX related deadlock in OpenMPI 4.1.2 (see flux-framework/flux-core#5460.)

latest

How should I configure OpenMPI to work with Flux?

There are many ways to configure OpenMPI, but a few configure options deserve special mention if MPI programs are to be run by Flux:

enable-static

> One of the Flux MCA plugins uses `dlopen()` internally to access Flux's `libpmi.so` library, since unlike the MPICH-derivatives, OpenMPI does not have a built-in simple PMI client. This option prevents OpenMPI from using `dlopen()` so that MCA plugin will not be built. Do not use.

with-flux-pmi

> Although the Flux MCA plugins are built by default, this is required to ensure configure fails if they cannot be built for some reason.

How do I make OpenMPI print debugging output?

This is not a Flux question but it comes up often enough to mention here. You may set OpenMPI MCA parameters via the environment by prefixing the parameter with `OMPI_MCA_`. For example, to get verbose output from the Block Transfer Layer (BTL), set the `btl_base_verbose` parameter to an integer verbosity level, e.g.

```
$ flux run --env=OMPI_MCA_btl_base_verbose=99 -N2 -n4 ./hello
```

To list available MCA parameters containing the string `_verbose` use:

```
$ ompi_info -a | grep _verbose
```

How should I configure MVAPICH2 to work with Flux?

These configuration options are pertinent if MPI programs are to be run by Flux:

with-pm=hydra

> Select the built-in PMI-1 "simple" wire protocol client which matches the default PMI environment provided by Flux.

with-pm=slurm

> This disables the aforementioned PMI-1 client, even if hydra is also spec

latest

Note

It appears that `--with-pm=slurm` is not required to run MPI programs under Slurm, although it is unclear whether there is a performance impact under Slurm when this option is omitted.

Why is MPI_Init() failing/hanging?

If your MPI application is not advancing past `MPI_Init()`, there may be a problem with the PMI handshake which MPI uses to obtain process and networking information. To debug this, try getting a server side PMI protocol trace by running your job with `-o verbose=2`. A healthy MPICH PMI handshake looks something like this:

```
$ flux run -o verbose=2 -N2 ./hello
0.731s: flux-shell[1]: DEBUG: 1: tasks [1] on cores 0-3
0.739s: flux-shell[1]: DEBUG: Loading /usr/local/etc/flux/shell/initrc.lua
0.744s: flux-shell[1]: TRACE: Successfully loaded flux.shell module
0.744s: flux-shell[1]: TRACE: trying to load /usr/local/etc/flux/shell/initrc.lua
0.757s: flux-shell[1]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/intel_mpi.lua
0.758s: flux-shell[1]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/mvapich.lua
0.782s: flux-shell[1]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/openmpi.lua
0.906s: flux-shell[1]: DEBUG: libpals: jobtap plugin not loaded: disabling
operation
0.721s: flux-shell[0]: DEBUG: 0: task_count=2 slot_count=2 cores_per_slot=1
slots_per_node=1
0.722s: flux-shell[0]: DEBUG: 0: tasks [0] on cores 0-3
0.730s: flux-shell[0]: DEBUG: Loading /usr/local/etc/flux/shell/initrc.lua
0.739s: flux-shell[0]: TRACE: Successfully loaded flux.shell module
0.739s: flux-shell[0]: TRACE: trying to load /usr/local/etc/flux/shell/initrc.lua
0.753s: flux-shell[0]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/intel_mpi.lua
0.758s: flux-shell[0]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/mvapich.lua
0.784s: flux-shell[0]: TRACE: trying to load
/usr/local/etc/flux/shell/lua.d/openmpi.lua
0.792s: flux-shell[0]: DEBUG: output: batch timeout = 0.500s
0.921s: flux-shell[0]: DEBUG: libpals: jobtap plugin not loaded: disabling
operation
1.054s: flux-shell[0]: TRACE: pmi: 0: C: cmd=init pmi_version=1 pmi_subversion=1
1.054s: flux-shell[0]: TRACE: pmi: 0: S: cmd=response_to_init rc=0 pmi_version=1
pmi_subversion=1
1.054s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get_maxes
1.054s: flux-shell[0]: TRACE: pmi: 0: S: cmd=maxes rc=0 kvsname_
keylen_max=64 vallen_max=1024
1.055s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get_appnum
1.055s: flux-shell[0]: TRACE: pmi: 0: S: cmd=appnum rc=0 appnum=0
```

```
1.055s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get_my_kvsname
1.055s: flux-shell[0]: TRACE: pmi: 0: S: cmd=my_kvsname rc=0 kvsname=ƒABRxM89qL3
1.055s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get kvsname=ƒABRxM89qL3
key=PMI_process_mapping
1.055s: flux-shell[0]: TRACE: pmi: 0: S: cmd=get_result rc=0 value=(vector,
(0,2,1))
1.056s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get_my_kvsname
1.056s: flux-shell[0]: TRACE: pmi: 0: S: cmd=my_kvsname rc=0 kvsname=ƒABRxM89qL3
1.059s: flux-shell[0]: TRACE: pmi: 0: C: cmd=put kvsname=ƒABRxM89qL3 key=P0-
businesscard value=description#picl6$port#41401$ifname#192.168.88.251$
1.059s: flux-shell[0]: TRACE: pmi: 0: S: cmd=put_result rc=0
1.060s: flux-shell[0]: TRACE: pmi: 0: C: cmd=barrier_in
1.059s: flux-shell[1]: TRACE: pmi: 1: C: cmd=init pmi_version=1 pmi_subversion=1
1.059s: flux-shell[1]: TRACE: pmi: 1: S: cmd=response_to_init rc=0 pmi_version=1
pmi_subversion=1
1.060s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get_maxes
1.060s: flux-shell[1]: TRACE: pmi: 1: S: cmd=maxes rc=0 kvsname_max=64
keylen_max=64 vallen_max=1024
1.060s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get_appnum
1.060s: flux-shell[1]: TRACE: pmi: 1: S: cmd=appnum rc=0 appnum=0
1.060s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get_my_kvsname
1.060s: flux-shell[1]: TRACE: pmi: 1: S: cmd=my_kvsname rc=0 kvsname=ƒABRxM89qL3
1.061s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get kvsname=ƒABRxM89qL3
key=PMI_process_mapping
1.061s: flux-shell[1]: TRACE: pmi: 1: S: cmd=get_result rc=0 value=(vector,
(0,2,1))
1.062s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get_my_kvsname
1.062s: flux-shell[1]: TRACE: pmi: 1: S: cmd=my_kvsname rc=0 kvsname=ƒABRxM89qL3
1.065s: flux-shell[1]: TRACE: pmi: 1: C: cmd=put kvsname=ƒABRxM89qL3 key=P1-
businesscard value=description#picl7$port#35977$ifname#192.168.88.250$
1.065s: flux-shell[1]: TRACE: pmi: 1: S: cmd=put_result rc=0
1.065s: flux-shell[1]: TRACE: pmi: 1: C: cmd=barrier_in
1.069s: flux-shell[1]: TRACE: pmi: 1: S: cmd=barrier_out rc=0
1.066s: flux-shell[0]: TRACE: pmi: 0: S: cmd=barrier_out rc=0
1.084s: flux-shell[0]: TRACE: pmi: 0: C: cmd=get kvsname=ƒABRxM89qL3 key=P1-
businesscard
1.084s: flux-shell[0]: TRACE: pmi: 0: S: cmd=get_result rc=0
value=description#picl7$port#35977$ifname#192.168.88.250$
1.093s: flux-shell[0]: TRACE: pmi: 0: C: cmd=finalize
1.093s: flux-shell[0]: TRACE: pmi: 0: S: cmd=finalize_ack rc=0
1.093s: flux-shell[0]: TRACE: pmi: 0: S: pmi finalized
1.093s: flux-shell[0]: TRACE: pmi: 0: C: pmi EOF
1.089s: flux-shell[1]: TRACE: pmi: 1: C: cmd=get kvsname=ƒABRxM89qL3 key=P0-
businesscard
1.089s: flux-shell[1]: TRACE: pmi: 1: S: cmd=get_result rc=0
value=description#picl6$port#41401$ifname#192.168.88.251$
1.094s: flux-shell[1]: TRACE: pmi: 1: C: cmd=finalize
1.094s: flux-shell[1]: TRACE: pmi: 1: S: cmd=finalize_ack rc=0
1.094s: flux-shell[1]: TRACE: pmi: 1: S: pmi finalized
1.095s: flux-shell[1]: TRACE: pmi: 1: C: pmi EOF
1.099s: flux-shell[1]: DEBUG: task 1 complete status=0
1.107s: flux-shell[1]: DEBUG: exit 0
1.097s: flux-shell[0]: DEBUG: task 0 complete status=0
```

```
ƒABRxM89qL3: completed MPI_Init in 0.084s.  There are 2 tasks
ƒABRxM89qL3: completed first barrier in 0.008s
ƒABRxM89qL3: completed MPI_Finalize in 0.003s
1.116s: flux-shell[0]: DEBUG: exit 0
```

Flux Developer Questions

My message callback is not being run. How do I debug?

- Check the error codes from `flux_msg_handler_addvec`, `flux_register_service`, `flux_rpc_get`, etc
- Use `FLUX_O_TRACE` and `FLUX_HANDLE_TRACE` to see messages moving through the overlay
- `FLUX_HANDLE_TRACE` is set when starting a Flux instance: `FLUX_HANDLE_TRACE=t flux start`
- `FLUX_O_TRACE` is passed as a flag to flux_open(3).

Apr 20, 2025

latest