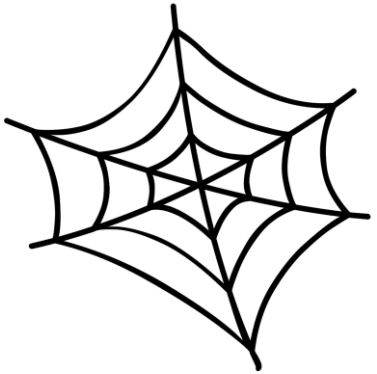
MOCHI BOOT CAMP

Argonne NATIONAL LABORATORY

SESSION 6: NETWORK FABRICS & RPC TUNING



PHILIP CARNS
Argonne National Laboratory

October 24-26, 2019

NETWORK FABRIC SUPPORT IN MOCHI

MERCURY NETWORK SUPPORT

"NA" plugins interface with communication libraries

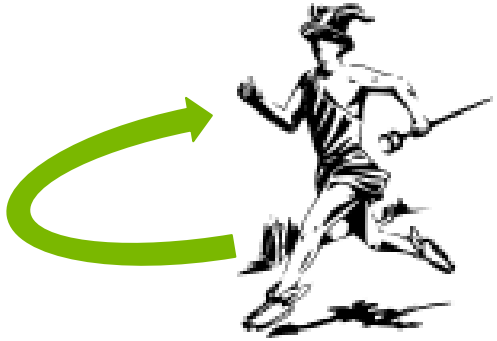
- **Libfabric**
  - Most important library for remote communication
  - Actively supported and improving
  - Supports many networks

Thank you to Intel for contributing the libfabric driver in Mercury.

- **NA SM**
  - Mercury's own transport for shared memory, on-node communication
  - Automatically used when "remote" peers are actually reachable on local node
- **Self RPCs**
  - Not really a transport, but a fast execution mechanism for in-process RPCs
- MPI: strictly for prototyping, not performant
- BMI: legacy code, not recommended

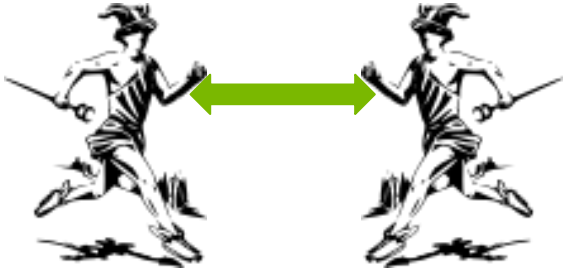
MERCURY “SELF” RPCS

Not really a transport, but good to know about!



- What if a process sends an RPC to itself?
- Mercury detects this and skips all NA plugins
  - RPC handler functions are invoked directly
  - Still delegated to the same Margo thread pool they would have been
  - Bulk transfers become memory copies
- Why is this important?
  - Flexible composition!
  - Components can communicate with other components using RPCs, regardless of whether peer component is in-process, on-node, or remote
- This functionality is enabled by default

MERCURY NA SM PLUGIN



- Can be selected explicitly using the “na+sm://” network address prefix, as we have done in some of the hands-on exercises
- Will also be implicitly used when a remote process is detected locally
- On Linux:
  - Uses Cross-Memory Attach capability (single copy)
  - If bulk transfers fail, on some system you may need (as root): “`echo 0 > /proc/sys/kernel/yama/ptrace_scope`”
- Others:
  - Uses conventional POSIX shared memory (2 copies)
- Enabled by default in spack or with NA_USE_SM variable in cmake

MERCURY LIBFABRIC: IB/VERBS

The essentials

- Address prefix: “verbs://”
  - Mercury will internally translate that to a more verbose “ofi+verbs;ofi_rxm://”
- Libfabric providers that will be used: “verbs” and “rxm”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/cooley
- packages.yaml:

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=verbs,rxm
```

MERCURY LIBFABRIC: IB/VERBS

Gotchas

- If using libfabric version >= 1.8: "`export FI_MR_CACHE_MAX_COUNT=0`"
  - Disables new memory registration cache, which has some performance flaws that impact Mochi (for now, hopefully fixed soon)
- If using libfabric versions < 1.8: "`export FI_FORK_UNSAFE=1`"
  - Mochi doesn't actually care about fork; this is a compatibility issue with MPI
- Aside from above caveats, works great! Fast and stable.
- Performance tuning: be aware that memory registration can be relatively expensive (margo_bulk_create()), especially for small transfers
  - We don't have a silver bullet yet, but for some size ranges we have better performance by copying through pre-pinned buffer rather than pinning in place

MERCURY LIBFABRIC: OMNIPATH

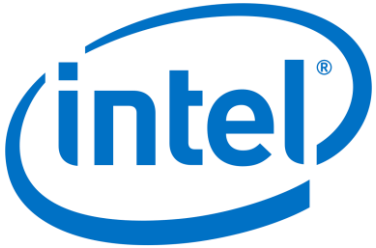
The essentials

- Address prefix: “psm2://”
  - Mercury will internally translate that to a more verbose “ofi+psm2://”
- Libfabric provider that will be used: “psm2”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/bebop
- packages.yaml:

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=psm2
```

MERCURY LIBFABRIC: OMNIPATH

Gotchas



- For any version: "`export FI_MR_CACHE_MAX_COUNT=0`"
  - Allows Mochi and MPI to share interface if needed
- Performance: excellent
- Stability: YMMV depending on system, library versions, and workload. We don't have universal solution to recommend yet.
- All OmniPath systems also support verbs through emulation, at a performance penalty. In the worst case, use verbs if you have to.
- Memory registration is free (noop), but not really because you pay performance cost at communication time on cold memory access.
- OmniPath/PSM2 consumes more host CPU than other high performance networks.

MERCURY LIBFABRIC: ARIES/GNI

The essentials

- Address prefix: “gni://”
  - Mercury will internally translate that to a more verbose “ofi+gni://”
- Libfabric provider that will be used: “gni”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/theta
- packages.yaml:

```
mercury:
      variants: ~boostsys+ofi
libfabric:
      variants: fabrics=gni
```

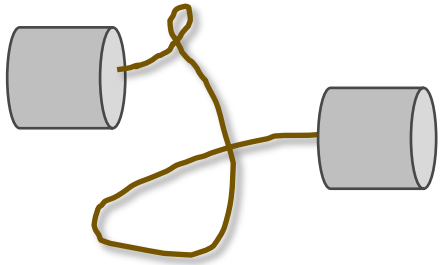
NOTE: This can only be built on a Cray machine, and you might need to load “ugni” module in your environment. Ugni is not a spack package.

MERCURY LIBFABRIC: ARIES/GNI

- "`export FI_MR_CACHE_MAX_COUNT=0`"
  - Tells Cray-MPICH to be less aggressive with resource consumption
- Communication across separately launched executables requires DRC, which in turn requires explicit setup within your code (external to Mercury)
  - DRC only works with libfabric 1.8.1rc or newer
- Stability is great
- Performance is (mostly) great but there are gotchas:
  - Latency is poor when not busy-spinning (more on how to do this later)
  - Latency is poor (any polling mode) on KNL cores
  - Memory registration / copy tradeoffs are unclear, possibly similar to IB/Verbs

MERCURY LIBFABRIC: TCP/IP

The essentials

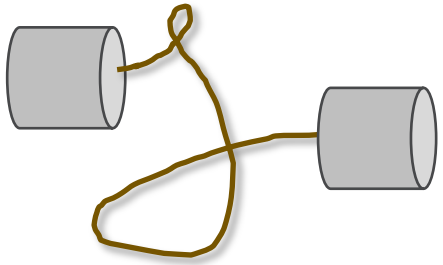


- Address prefix: “tcp://”
  - Mercury will internally translate that to a more verbose “ofi+tcp;ofi_rxm://”
- Libfabric providers that will be used: “tcp” and “rxm”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/cooley/*tcp*
-

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=tcp,rxm
```

NOTE: there are alternative TCP implementations, but this one is our recommendation.

MERCURY LIBFABRIC: TCP



- Please use libfabric version >= 1.8.0
- Stability is good
- Performance is poor (when other alternatives are possible)
  - This is largely because of emulation of RDMA over sockets
- There are other options that you may see in previous examples:
  - “sockets://” activates the legacy/reference sockets implementation in libfabric
  - “bmi+tcp://” activates TCP support in the BMI library (if enabled)
  - All have similar performance characteristics; “tcp;ofi_rxm” is now the most actively maintained, though

POLLING MODES (ANY TRANSPORT)

Hey network card: Are you done? Are you done? Are you done?

- Ok, how about now?
- Generally speaking, any of the network transports can be driven in “polling mode”, where a CPU core constantly checks for progress.
  - This is fast - no interrupt or context switch when messages complete
  - This also eats a CPU core. The value of this tradeoff depends on the use case
  - Benchmarks almost always busy poll.
- Margo defaults to an adaptive model: it idles gracefully when there is nothing to do, switches to busy mode under load.
- You can force it to always busy poll by setting progress_mode to NA_NO_BLOCK

https://xgitlab.cels.anl.gov/sds/sds-tests/blob/master/perf-regression/margo-p2p-latency.c#L91

THE STATE OF MOCHI TRANSPORTS

September 2019

- Things are continually improving, particularly in the interaction between Mochi, libfabric, and MPI
- There are some quirks.
- Want to keep up over time? Your best option right now is to monitor the scripts that we use for regression testing on different transports:

https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression

- We update these over time with best practice for platforms at ANL

RPC TUNING

BUILT-IN RPC PROFILING

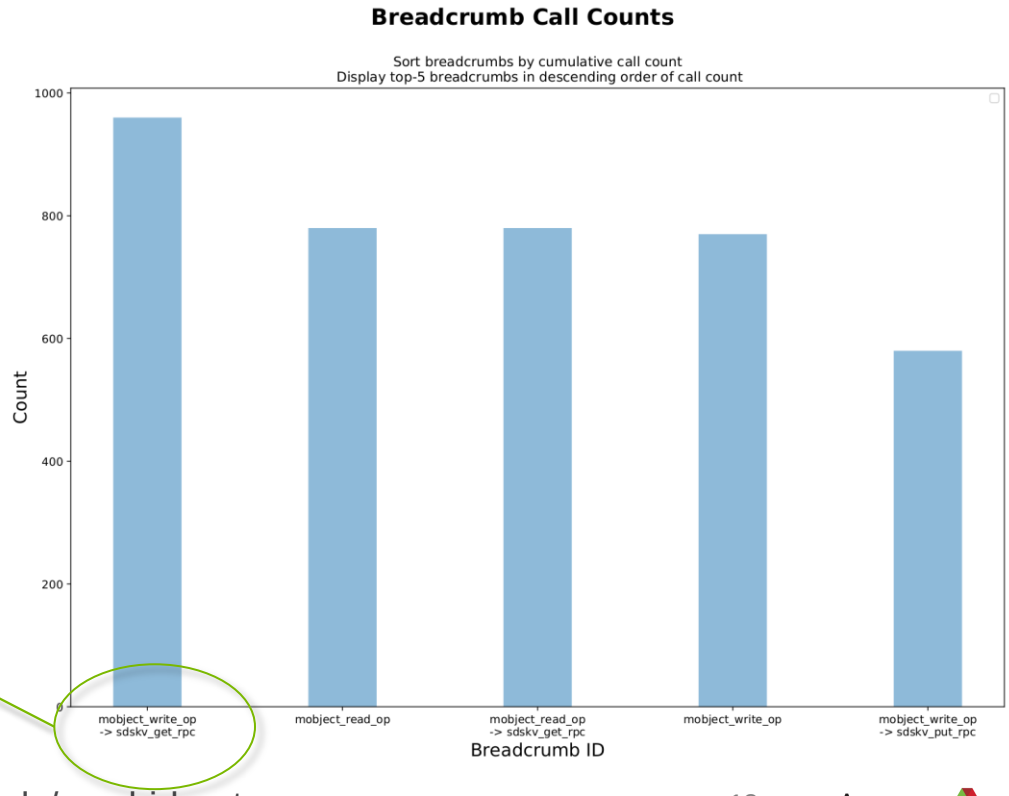
Coming soon!

- How do you tune a Mochi service, or at least understand it's performance?
- Almost every major operation in Mochi is an RPC
  - RPCs can be chained, local, remote, etc.
- If all RPCs in a service were transparently instrumented, then any Mochi service could be quickly characterized
- This is an upcoming feature, thanks in large part to work by Srinivasan Ramesh of the University of Oregon
- Usage is modeled after Darshan in many ways:
  - Characterization capability is compiled in, no code modification needed
  - Simple command line tools to produce a PDF with plots to use as a starting point for performance tuning

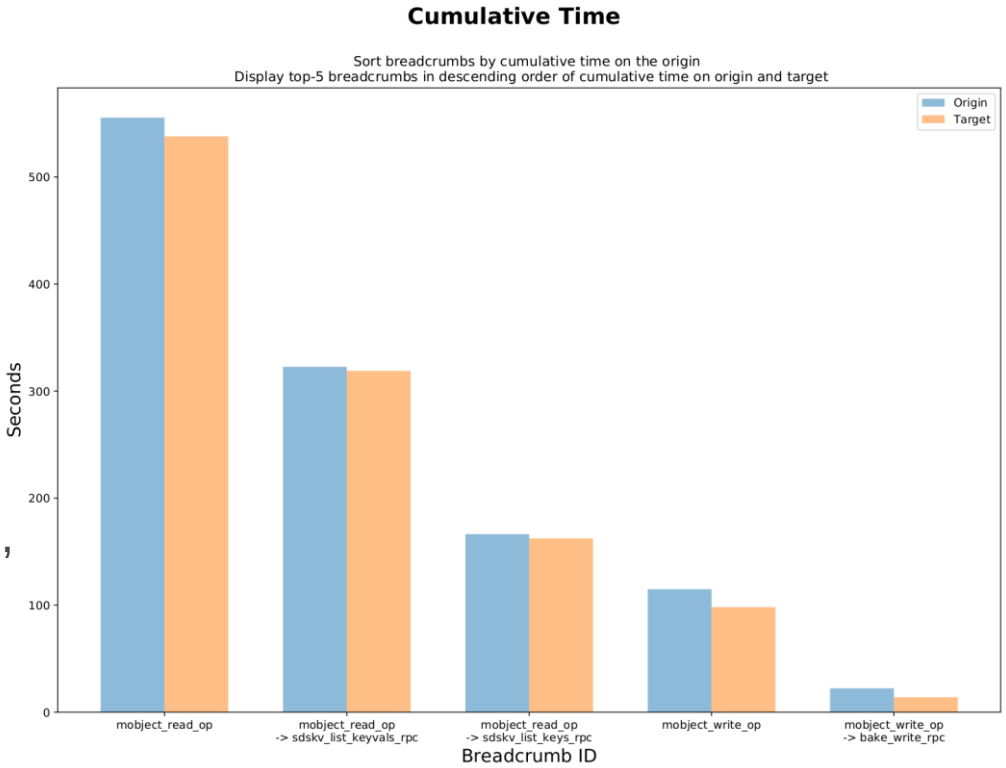
MOCHI RPC BREADCRUMBS

What RPCs are most prolific?

- This graph shows ranked list of 5 most frequently called RPCs
- Each includes lineage if applicable (the chain of RPCs that led to the one being measured)



- This graph shows ranked list of 5 RPCs that cumulatively consumed the most service time
- Split by client and server side elapsed time
- Each includes lineage if applicable, just as in the previous graph.



RPC BREADCRUMBS

Status

- Mochi already includes the mechanism for measuring RPC time and tracking the lineage of RPCs
  - The examples you run today include this capability already
- Remaining work is productizing the tools that can dump summary data and plot it
- There are many other possibilities for deeper analysis from that starting point

THANK YOU!