—dc={datacenter name}
—env={environment name}

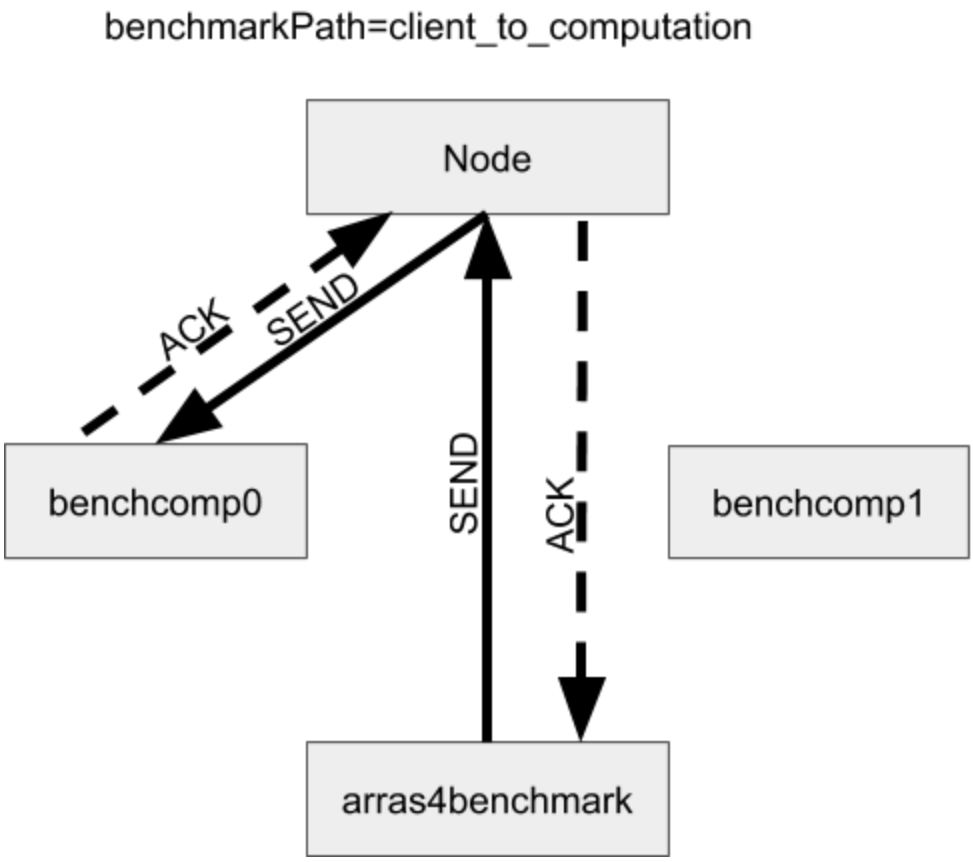
One model or specifying the url to connect to is using **dc** (which probably defaults to what you need) and **env** which defaults to prod and can at the time of this writing can be {prod, usn, stb, stb2}

—host={hostname}
—port={port #}

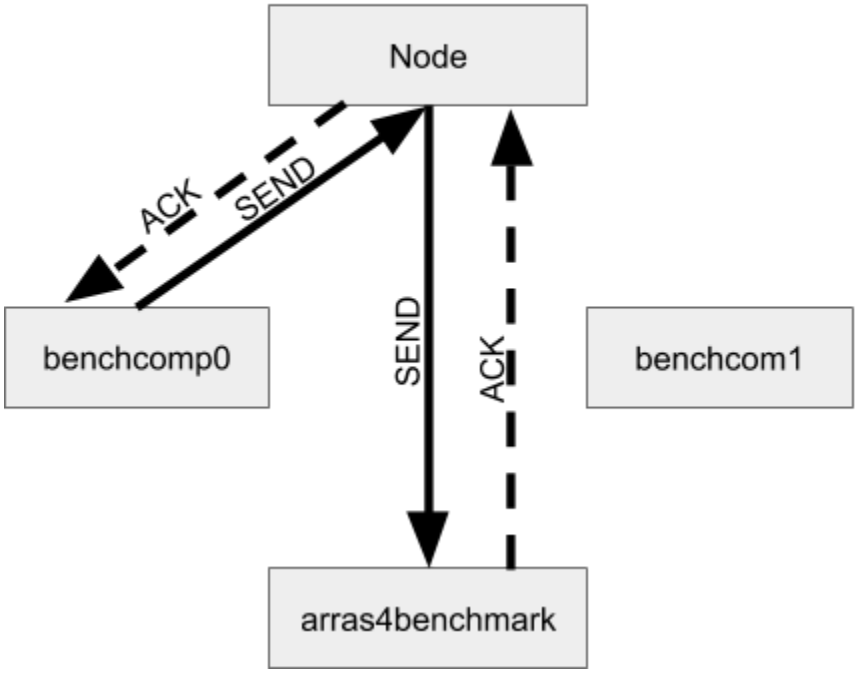
Another way to specifying the url to connect to is using **host** and **port.** The most common use of host and port will be for handling local environment in which case it would **host=localhost** and **port=8087**. It is conceivable that some testing would be wanted with a manually generated environment but running the client on another machine in which case the host would be

—benchmarkPath={client_to_computation, computation_to_client, computation_to_computation}

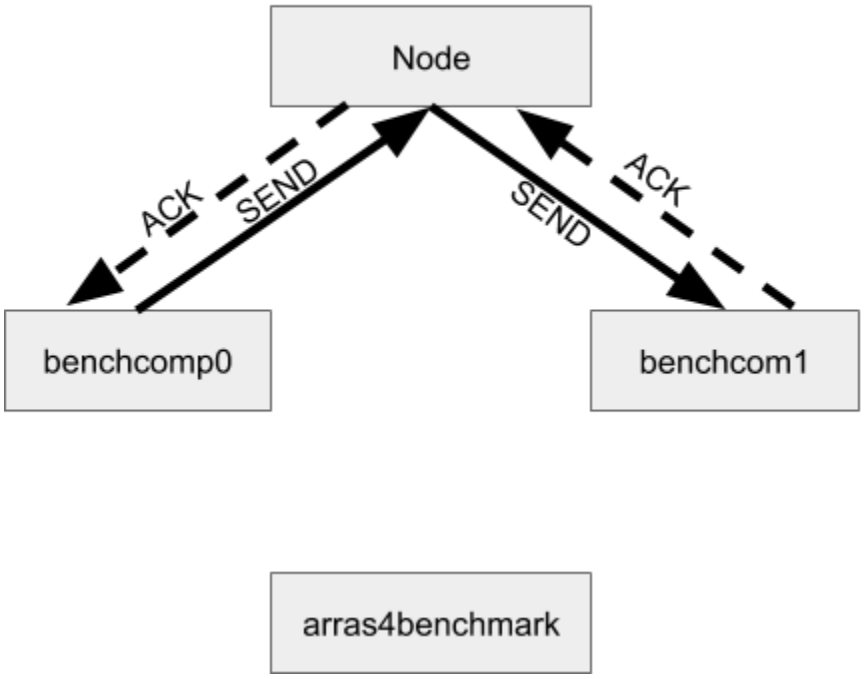
Test one of three paths for sending data as quickly as possible.From client to a computation (i.e. sending render data to mcrt), from a computation to the client (i.e. sending images back to the client), and between computations (mcrts sending samples to the merge computation).



benchmarkPath=computation_to_client



benchmarkPath=computation_to_computation



—mb={*number of megabytes*}

—bytes={*number of bytes*}

Specify the size of the individual messages being sent. The size will be (mb * 1048576 + bytes).
The expectation is that tiny messages would be dominated by software overhead while large messages would be limited by data transmission/copy speeds.

Tiny messages (i.e. bytes=1) transmit more data that just the one byte due to header overhead but this is generally a test for a message rate. Increasing the message rate will generally involve optimizing software to simply reduce the amount of work that is done routing messages or allowing the necessary work to happen in parallel.

Large messages are useful for finding maximum bandwidth. How large do the messages need to be to get the full bandwidth through the system? There might be a peak size beyond which performance goes back down due to poor cache usage. Increasing the bandwidth will generally involve using a better transmission mechanism, reducing the number of copies of the data that need to occur, and possibly allowing necessary work to happen in parallel.

—credits={*number of credits*}

Credits is the number of messages which can be sent without receiving an acknowledgement. Using credits=1 is completely synchronous communication and is a good way to measure latency through the system. Using credits=1000000000 would probably make the acknowledgement mechanism irrelevant and would be a way to see what might break in the system from sending data too fast. Does data back up in queues somewhere consuming too much or creating extreme latency?

—duration={*seconds*}

How long the test should run not counting the “stay-connected” time

—report-frequency={*seconds*}

How often to print an incremental report. Defaults to 5 seconds. Incremental reports can help show how much variation occurs or allows watching the bandwidth change as other things are changed.

—cores={# of cores}
—requestMb={# of megabytes}

How many cores and how much memory to ask for in resource requirements for the main computation. This can be used to watch the behavior or coordinator with different allocations or to test the enforcement or reporting of core usage.

—threads={# of threads}

How many threads to run consuming cpu resources on the main computation. This is allowed to be more than the number of requested cores by design to test enforcement of resource usage or to intentionally oversubscribe cores.

Having the system be busy could affect message transmission rates or bandwidth by reducing CPU resources available for message management or creating latency these startup due to contention over cores.

—allocateMb={# *of megabytes*}
—touchMb={# *of megabytes*}

How much memory should the main computation allocate and how much of that memory should actually be used? This can be used for testing the enforcement and reporting of memory. Allocation and

touching are separate because there are possibly separate mechanism for measuring and enforcing memory allocated but never touched and memory which is actually written. Depending on the rules in place Linux can allow more memory to be allocated than it available. This can also be used to create memory activity to possibly affect core caches which could affect messaging speeds.

—touchOnce

When writing memory only write it once rather than continuously. This cause the memory to be used without causing ongoing memory I/O or CPU usage.

—logthreads={# of threads that are printing, default 12}
—logCount={# of 80 character lines to log, default 0}

When logCount is set the specified number of threads will will each write an to stderr logCount/logThreads times stderr as quickly as possible with 80 character lines.

—printEnv

Dump out all of the environment variables in the computation's environment

—prepend={additional package directory}

Add a directory to the computation's rez environment to allow test maps to be used on pool machines.

Benchmarking Examples

Test basic message processing latency sending from client to computation by using tiny messages and only 1 credit to force a synchronous round trip for each message

$ arras4benchmark --host localhost --port 8087 --bandwidthPath=client_to_computation --bytes=1 --credits=1 --duration=30 --report-frequency=5
Time: 5.00s Msgs: 38876 Rate: 7775.15msg/s (128.61µs) 0.01MB/s TOTALS: Time: 5.00s Msgs: 38876 Rate: 7775.15msg/s (128.61µs) 0.01MB/s
Time: 5.00s Msgs: 40427 Rate: 8085.12msg/s (123.68µs) 0.01MB/s TOTALS: Time: 10.00s Msgs: 79303 Rate: 7930.14msg/s (126.10µs) 0.01MB/s
Time: 5.00s Msgs: 40877 Rate: 8175.19msg/s (122.32µs) 0.01MB/s TOTALS: Time: 15.00s Msgs: 120180 Rate: 8011.82msg/s (124.82µs) 0.01MB/s
Time: 5.00s Msgs: 35876 Rate: 7174.92msg/s (139.37µs) 0.01MB/s TOTALS: Time: 20.00s Msgs: 156056 Rate: 7802.59msg/s (128.16µs) 0.01MB/s
Time: 5.00s Msgs: 36110 Rate: 7221.73msg/s (138.47µs) 0.01MB/s TOTALS: Time: 25.00s Msgs: 192166 Rate: 7686.42msg/s (130.10µs) 0.01MB/s
Time: 5.00s Msgs: 37245 Rate: 7449.00msg/s (134.25µs) 0.01MB/s TOTALS: Time: 30.00s Msgs: 229411 Rate: 7646.85msg/s **(130.77µs)** 0.01MB/s

Allowing some asynchronous behavior by letting 5 messages be outstanding doubles throughput for these small messages

$ arras4benchmark --host localhost --port 8087 --bandwidthPath=client_to_computation --bytes=1 --credits=5 --duration=15 --report-frequency=15
Time: 15.00s Msgs: 225236 Rate: 15015.72msg/s **(66.60µs)** 0.01MB/s TOTALS: Time: 15.00s Msgs: 225236 Rate: 15015.72msg/s (66.60µs) 0.01MB/s

There is a slight benefit for 10 credits but no real value to 100 over 10 so it's reached the point of diminishing returns.

$ arras4benchmark --host localhost --port 8087 --bandwidthPath=client_to_computation --bytes=1 --credits=10 --duration=15 --report-frequency=15
Time: 15.00s Msgs: 240751 Rate: 16050.02msg/s **(62.31µs)** 0.02MB/s TOTALS: Time: 15.00s Msgs: 240751 Rate: 16050.02msg/s (62.31µs) 0.02MB/s
]$ arras4benchmark --host localhost --port 8087 --bandwidthPath=client_to_computation --bytes=1 --credits=100 --duration=15 --report-frequency=15
Time: 15.00s Msgs: 242659 Rate: 16177.26msg/s **(61.82µs)** 0.02MB/s TOTALS: Time: 15.00s Msgs: 242659 Rate: 16177.26msg/s (61.82µs) 0.02MB/s

Running with different sizes of messages we see the bandwidth (in MB/sec) increase as the messages get larger up to a megabyte then drop off as the messages continue to get larger. Some resource is likely being blown out making things less efficient.