From: Ahn, Dong H.
Sent: Friday, July 19, 2013 6:26 PM
To: flux-discuss@lists.llnl.gov
Cc: Ahn, Dong H.; Moody, Adam T.; Legendre, Matthew P.; Lee, Greg
Subject: The baseline performance of COBO on CMB

All:

In the past few weeks, I've ported COBO (a bootstrapper library for distributed HPC software like run-time tools and middleware like LaunchMON, STAT, SPINDLE, MRNet etc) to CMB. And before moving onto my next set of tasks, I've measured its baseline performance on CMB today.

I started from the top of PMGR stash repo and added COBO interfaces on top of that, and then I've added CMB KVS support to this version of COBO. With CMB KVS support, each of the launched COBO back-end processes initially puts and commits its IP and an ephemeral port of a listening socket, using its rank as the key. Then, during forming a TCP tree overlay network, each process in the tree gets the IP/port pair of its parent, using the parent's rank as the key. (I shouldn't take too much credit for this because this scheme is simply a replication of PMGR's PMI support.)

I compared the cobo_open overhead of this version w/ that of the original scheme, which connects all back-end processes into a tree. I call the original scheme as "connect-forward." All COBO back-end processes are spawned first w/ no rank assigned to them, and each of them subsequently opens a port in a port range and waits for a connection. Then, the root process initiates the connection. It determines the host of each of its children and attempts to connect to a port in the port range until the connection succeeds (with some handshaking protocol). Once the connection is complete, the entire host list gets transferred to the children processes which then perform the same connect-forward scheme. This is recursively done until they reach leaf processes.

The attached MS spreadsheet (COBO-CMB.xlsx) has the data points and graphs. I ran both vanilla COBO and CMB COBO testers at 32, 64 and 100 hype nodes. In particular, I ran each tester essentially three times at each node count with increasing overcommit factors: 16 processes per node (i.e., 1x overcommit factor), 32 processes (i.e., 2x overcommit) and 64 processes (i.e., 4x overcommit). E.g., at 100 nodes, I ran each tester with 1600, 3200 and 6400 back-end processes. And each data point is the average of three different runs, and CMB is configured w/ the binary topology. The spreadsheet has a separate tab for each overcommit factor.

Looking at the graphs, it seems the original scheme performs a bit better than COBO/CMB. Although inconclusive, it seems there is a constant overhead w/ COBO/CMB on top of the original COBO. Because the used version of Adam's PMGR has some initial checking code that isn't present in the original COBO, I suspect this constant overhead can be explained by them. I didn't bother to gather a further breakdown on where the time is going because the point of this exercise was merely to do some sanity check by scaling up this initial port.

At some point, I will want to scale this further up and measure some more fine-grain metrics. But I will wait until we place more key components such as remote execution mechanism (which I plan to work w/ Mark next week) and more importantly distributed key-vale store support.

I will commit this code base to our github repo soon so that folks can play with it if they want.

My planned tasks:

- LaunchMON/CMB path-forward design (3 days)
- LaunchMON 1.0 support natively on CMB (three weeks)
- LIBI support natively on CMB -- evaluation/port (7 days)
- MRNet port on top of LIBI (2 weeks)

Once these basic interfaces are ported, I should be able to start to port tools/middleware like STAT and SPINDLE to FLUX environment, and show that its run-time is rich to support various types of lightweight jobs and also highly scalable. I have some ideas to show what this approach does much better than other RM's run-time environments.

Best,
Dong