Running a Monte-Carlo simulation on a Quantlab cluster

This is an example of how to create a mini-cluster on separate servers/desktop pc:s or locally and then executing a distributed call to price an Asian option using Monte-Carlo simulation.

We will assume that all servers that should run a Quantlab instance will have a valid licence key and a proper stand-alone Quantlab installed somewhere.

In the example, an alternate call to multiple threads within the same physical multi-core server will also be presented. When running locally, the QLC service(s) are not needed.

To install the QLC, the Quantlab Calc Server, simply write quantlab –S <service_name> <description> from the command prompt on the server that should have the QLC running as a service.

For all the QLC services installed on the cluster, log-on and start them using the sevices interface in the usual manner. We are now ready to create a distributed calculation problem.

Note! that this is a simplistic example where the cluster nodes do not have to be aware of each other. When there is need for intra-node communication, a cluster should be initiated using the qlc.create_cluster function. Then all the usual cluster node controls will be available such as rank, size, send, receive, distribute, gather, and broadcast.

Coding and distribution

This example is a simplistic Monte-Carlo simulation to get the price of a path-dependent (arithmetic average) European option.

The idea is to treat the total number of simulations as a number of independent simulations carried out on n-servers, each running total / n simulations. When all servers are finished we simply average over the n-result sets. In order for this simple trick to work we need to ensure that the statistical properties will not be changed.

To ensure that we have a unique run of random numbers we can seed the random number generator on each server with a different starting seed. Here we will use the internal clock's milliseconds to seed the random number function. An alternative would be to fixate the starting seeds in order to have a reproducible result.

There are two ways of getting the function to execute onto the remote servers. You can install the function locally on each server as a library file (which would be the normal case). But you may also send a string containing the complete code at run-time. It will then be compiled when the connection to the remote server is done. In this example we use this option for simplicity.

Let's look at the code:

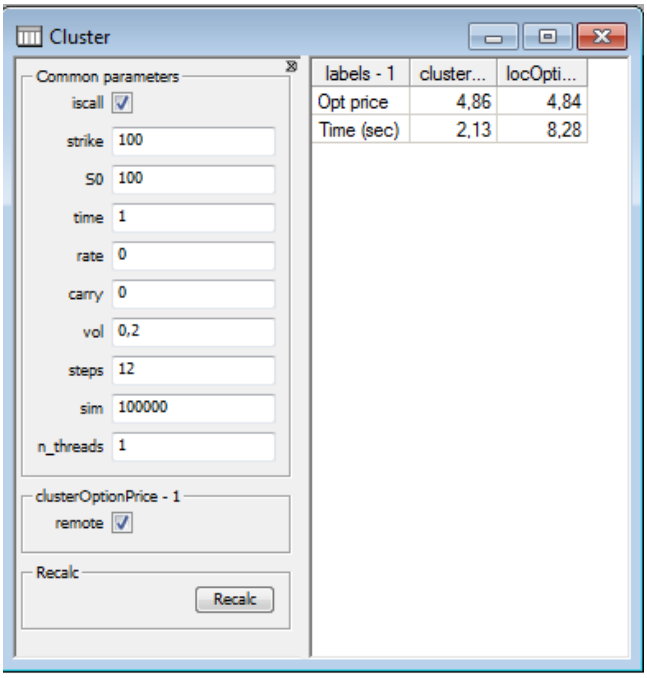
```
//First we list all our servers by name - here we have 4 qlc in place
vector(string) v_servers = ['lincoln','jefferson', 'roosevelt','washington'] ;

//Second we concatenate a string with the simulation code function
string myQlang = strcat([
```

```
            'out number myoptionpayoff(logical iscall, number strike, ',
                        'number S0, number time, ',
                        'number rate, number carry, number vol, ',
                        'number steps, number sim){ ',
                        'rng r = rng(millisecond(now())); ',
                        'number dt = time / steps; ',
                        'number drift = (carry - vol*vol / 2) * dt; ',
                        'number dt vol sqr = vol * sqrt(dt); number z; ',
                        'if(iscall){ z=1 ;} else {z=-1;} ',
                        'number Payoffsum = 0;',
                        'for(number s = 1; s <=sim; s++){ ',
                        'number St = S0; ',
                        'number Ssum = 0; ',
                        'for(number j = 1; j <=steps; j++)       { ',
                        'St = St * exp(drift + dt vol sqr * r.gauss());',
                        'Ssum = Ssum + St; } ',
                        'number Savg = Ssum / steps;',
                        'Payoffsum = Payoffsum + max(z * (Savg - strike), 0); }',
                        'return exp(-rate * time)*Payoffsum/sim;}']);


//Here is the function that will assign the call to the cluster
//It has a number of input arguments so that the user can change the settings

out vector(number) clusterOptionPrice(logical iscall, number strike, number S0, number time,
number rate, number carry, number vol, number steps, number sim)
{
            timestamp start = now() ; //To time our tests

            //we connect to the servers and pass the function as a string argument
            vector(qlc.unit) v conn =  qlc.create thread remote(v servers, myQlang);
            number localsim = sim/v size(v servers);

            //using our vector of connections, we start the evaluation of the function
            //we pass the input arguments as variants to the remote servers
            v conn.start eval('myoptionpayoff',[qlc.variant(iscall), qlc.variant(strike),
               qlc.variant(S0), qlc.variant(time), qlc.variant(rate), qlc.variant(carry),
               qlc.variant(vol), qlc.variant(steps), qlc.variant(localsim)]);

            //now it's just to wait for the servers to finish and sum the results
            //the return is also a variant so we have to use the get_number()
            number ret = v sum(v conn.wait eval().get number())/v size(v servers);

            timestamp stop = now() ;
            number duration = (stop - start)/1000 ;
            return [ret,duration];
}
```

If running locally, the only difference is that we initiate the vector of qlc.units using a given number of calculation threads. Then the code would look like this:

```
            vector(qlc.unit) v_conn;
            for (number k = 0; k<n_threads;k++)
                 push_back(v_conn,qlc.create_thread(myQlang));
```

In order for us to get a user interface we attach the clusterOptionPrice function in table. The input arguments appear as controls. For comparative reasons we have also taken an identical version of the function that run locally but only using a single calculation thread, in order to get a sense of the speed gains from the cluster.



Here we are pricing an average option with 1 year maturity and monthly averaging points. We run 100000 simulations in total having 12 steps per simulation. The performance gain of using the four pc cluster is of course of magnitude 4:1 compared with running it on the single thread.

Can we do even better?

Actually yes! Since we know that the remote servers are running a dual core processor we can just add another call to the same servers in our v_servers vector. This will give us an additional calculation thread per server. Some performance is of course lost in overhead.

```
//First we list all our servers by name - now we get two threads on each server
vector(string) v_servers = ['lincoln','jefferson', 'roosevelt','washington',
                            'lincoln','jefferson', 'roosevelt','washington'] ;
```