Home
Gallery
Code

Parallel computing in IDL

Version 6.3 of IDL introduced the IDL-to-IDL bridge. Basically, this is a mechanism to start IDL sessions within IDL and communicate with them. Unfortunately, using it for parallel computing in IDL is rather more involved than one might like.

Limitation

The IDL-to-IDL bridge requires an X11 connection. So much for running some huge calculation on a remote machine in a screen session.

Building Blocks

Initializing

IDL-to-IDL bridges are created through the command:

```
bridge = obj_new('IDL_IDLBridge')
```

Unfortunately, the IDL session they create doesn't execute your `$IDL_STARTUP`, nor does it change its working directory to where you think it would be. You have to do that manually:

```
bridge->execute, '@' + pref_get('IDL_STARTUP')
bridge->execute, "cd, '" + pwd + "'"
```

I created a wrapper, `build_bridges` that creates a given number of bridges (defaults to the number of CPUs in the system) and for each one does the above. I also change the number of threads each bridge may run to 1 or a number specified by the user.

Using the Bridge

We can now use the bridges to do some work. Of course, we want to capture the output, and perhaps we want to pass some input.

```
bridges = build_bridges()
in = indgen(10,20)
pout = ptr_new(fltarr(10,20), /no_copy)
```

We'll use a callback routine to populate the output array with the results.

```
pro callback, status, error, bridge, ud
        out = bridge->getvar('out')
        (*(ud.pout))[ud.i,ud.j] = out
end
```

The bridge needs to know it has to call the callback routine after it completes a task.

```
bridge->setproperty, callback='callback'
```

Then, we can loop over all work packages. We have to set up the 'userdata' structure `ud` appropriately for the callback.

```
for i=0l,9 do for j=0l,19 do begin
        ud = {i:i,j:j,pout:pout}
        bridge = get_idle_bridge(bridges)
        bridge->setproperty, userdata=ud
        bridge->setvar, 'in', in[i,j]
        bridge->execute, /nowait, 'worker, in, out'
endfor
barrier_bridges, bridges
```

There are two routines used that I haven't introduced yet. `get_idle_bridge` is a wrapper that returns an `IDL_IDLBridge` object of a bridge that is currently idle. It blocks until it finds one. `barrier_bridges` blocks until all bridges are idle, i.e., all bridges have finished their computations.

At this point, our program `worker` has processed all inputs, and the output is available to our interactive session:

```
out = (*pout)
ptr_free, pout
```

Cleaning up

After we're done with our bridges, we can destroy them:

```
burn_bridges, bridges
```

Of course, if you have more calculations to do later, you don't have to destroy the bridges. They'll just be idle until you need them again.

Download

Download my IDL-to-IDL bridges wrappers.

Changelog

2011 04 22 initial release

Bugs & feature requests

Please send bugreports and feature requests to `dwijn` at `iluvatar.org`.