DEBUGGING EXAMPLES

Oh no! The VH1 example has developed three nasty bugs and only you can fix them.

```
 ____  _   _  ____
| __ )| | | |/ ___|
|  _ \| | | | |  _
| |_) | |_| | |_| |
|____/ \___/ \____|

                          )  (
                        _(.--.)
  _ _ _ _ _ _ _ _ _ _ _ ( '_')
 {{ { { { { { { { { { { ( '_')
  >>>>>>>>>>>>>>>>>>>>>>`--'>
```

Try and use some to the tools described in this document to fix the issues that you're seeing with the application.

Some things you might need to remember when staring off:

- VH1 requires the `cray-netcdf` module be loaded to build.
- You don't need perftools so unload it if you have it loaded from a previous exercise.
- You might also want to change the `Makefile` so that compilation includes debugging. E.g. for the Cray compiler, in `src/Makefile` change FFLAGS_CRAY = -O2 –emf to FFLAGS_CRAY = -g –emf, then do make clean followed by make.

In no particular order:

ATP (ABNORMAL TERMINATION PROCESSING)

FOR WHEN THINGS GO WRONG UNEXPECTEDLY

ATP kicks in when an application issues a signal indicating it has encountered a problem. This can range from a segmentation violation to a termination signal indicating that the application has run to long. ATP requires the environment variable

```
export ATP_ENABLED=1
```

To be set in the run script before it can become active.

Once activated and triggered via a signal, ATP will attempt to provide three types of diagnostic information to the user:

1. A backtrace of the PE that first generated the signal detected by ATP. This is printed to `stderr` so can usually be found in either the PBS error file, or the PBS output file output has been joined.
2. A merged back trace that provides a succinct visual representation of all the backtraces from all ranks in an application at failure. These are left in the working directory of the run and are killed:
    a. `atpMergedBT.dot` – Backtraces merged by function name only (smaller number of branches)
    b. `atpMergedBT_line.dot` – Backtraces merged by function name and line number (more accurate but larger trace)

    These can be visualised using the `stat-view` visualisation tool (though in essence they are standard GraphViz DOT files). To this run:

```
module load stat
stat-view atpMergeBT.dot
```

3. A selection of core files, each one representing a PE which is a member of one of the equivalence sets of the merged backtrace. i.e. a PE that is in a different terminal leaf of the merged backtrace tree. Files will be uniquely identified to a specific run and name via their name in the form `core.atp.<apid>.<PE>`.
    Remember, however, that core files will only be created if they have been enabled in the batch run script, this can be done by adding the following line before the `aprun` command.

```
ulimit –c unlimited
```

GETTING INTERACTIVE WITH PBS

Most Cray XC30s are configured not to allow directive interactive, on-demand access to their compute nodes. Instead resources must be requested via a batch scheduling system. In most cases this involves creating a run script that will be executed once the scheduler provides the required resources, however PBS allows users to request an interactive session instead.

This interactive session comes in the form of a direct connection with the PBS MOM node and allows users to interactively issue commands in lieu of submitting a script, perfect for debugging applications when you need quick turn around when resubmitting the same parallel application.

To request an interactive session, pass the standard resource requests to a `qsub` command, but in addition add `–I`, e.g.

```
qsub -l EC_nodes=4 -l walltime=0:20:00 -q np -I
```

Once an interactive session has been requested, the job enters the batch system just like any other job and must wait until its resources are allocated and the `qsub` command will block after printing the message:

```
qsub: waiting for job 127743.sdb to start
```

When the resources become available the following message will appear

```
qsub: job 127743.sdb ready
```

And a new interactive login shell will be spawned. Users can then issue command in this interactive shell just as they would a normal shell with the exception that requests to aprun will succeed as there are dedicated compute nodes available for use.

Warning: The time between job submission and resources becoming available can potentially very long. It is very easy for users to forget they have pending sessions and end up paying for large amounts resource that goes unused! Consider setting an appropriate walltime limit.

Once you have an Interactive login session you can use it for debugging with:

STAT: FOR WHEN THINGS APPEAR TO BE HANGING

The Stack Trace Analysis Tool (STAT) is the parent tool of ATP, it was designed to provide a snapshot back trace of an entire application, specifically when it appears to be in deadlock or hung for some reason. It can be activated either as part of a batch job, or as via of an interactive session (see Getting interactive with PBS above for more information on starting interactive sessions).

Batch STAT

If your application reliably enters a hung state then STAT can be incorporated into the batch script to initiate a whole application backtrace at a set time interval. This is done by wrapping the aprun launch command with the STAT executable after the stat module has been loaded. e.g.

Module load stat

```
stat-cl –s 30 –C aprun –n 24 –N24 vh1-cray-exe
```

The –s <n> option indicates that STAT should wait <n> seconds and then attempt to perform the snapshot. Therefore this value should be sufficiently large that the application can be reliably assumed to be in the hung state.

Two new directories are then created underneath one-another in the working directory:

```
stat_results/<exe>.<snapshot>/
```

Inside there will be a snapshot file

```
<exe>.0000.3D.dot
```

Which can be rendered using the `stat-view` program

```
module load stat; stat-view <exe>.0000.3D.dot
```

Interactive STAT

If your job does not hang repeatedly and hence you cannot use a timer to take a snapshot it is possible to use an interactive shell to run your script.

First, request an interactive session from the PBS scheduler, when that is ready run

Once the job has started, you can launch your standard run script in the background, e.g.

```
run_ECMWF.pbs &
```

However you may need to set any EC_ environment variables that your script needs. Once the job is running you need to identify the PID of the aprun command launch command, e.g.

```
ps aux | grep $USER | grep aprun
```

This may produce two aprun commands, e.g.

```
tedwards 13771  0.1  0.0  47600  2388 pts/5    tl   10:02   0:00 aprun -n 24 -N 24 ./vh1-mpi-cray
tedwards 13779  0.4  0.0  47336  1396 pts/5    S    10:02   0:00 aprun -n 24 -N 24 ./vh1-mpi-cray
```

Select the larger of the two PIDs and use this for `STAT`.

```
stat-cl <pid>
```

We would recommend running multiple times, producing multiple snapshots, incrementing the value under the stat_results file. They can be viewed in the same way using statview.

LGDB: THE COMMAND-LINE PARALLEL DEBUGGER

Before going any further, it is recommended to rebuild the executable `vh1-mpi-cray` to include debugging information. (Please see note on page 1.)

To access LGDB load the module at the command line

```
module load cray-lgdb
```

Attaching to a running program

Find the `apid` of the running process using `apstat`

```
 Apid ResId     User  PEs Nodes   Age State      Command
27349 12089 tedwards   24     1 0h00m   run vh1-mpi-cray
```

Launch `lgdb` with:

```
lgdb
```

And from the lgdb shell attach to the program as follows

```
attach $<pset> <apid>
```

Launching a new application

Debugging with lgdb may be achieved either by using an interactive session from PBS or by submitting a slightly modified PBS script from a batch session. To request a new interactive session from PBS (See Getting Interactive with PBS) proceed as follows.

Launch `lgdb` with:

```
lgdb
```

And from the `lgdb` shell run

```
launch $pset{nprocs} <exe> --aprun-args=”-N 24”
```

to launch using nprocs processes and the additional aprun arguments provided.

To use lgdb without the need for an interactive session, firstly, modify your existing PBS script by placing the comments `#cray_debug_start` and `#cray_debug_end` around the aprun command that is to be debugged. Everything between the comment lines will be ignored; the aprun command must then be recreated through the launch command arguments within lgdb. For example, the job script `sample.pbs` is modified with the lgdb comments:

```
#cray_debug_start
aprun -n128 -N32 a.out
#cray_debug_end
```

The following launch command will then submit `sample.pbs` as a PBS job from within which the application `a.out` will be launched:

```
dgb all> launch $a{128} --aprun-args="-N32" --qsub=sample.pbs a.out
```

NB: for the example, if you are debugging with lgdb with an interactive session, then please be careful to ensure that the files `indat`, `vh1-mpi-cray`, and a directory called `output` are all present in the current directory before executing `launch` (look at the original .pbs file to check), e.g. from within the interactive session:

```
RUNDIR=${ROOTDIR}/${PBS_JOBID}
BINDIR=${ROOTDIR}/../bin
# Create the output directory
mkdir -p ${RUNDIR}/
mkdir -p ${RUNDIR}/output
cd ${RUNDIR}
cp ${ROOTDIR}/indat ${RUNDIR}
cp ${BINDIR}/${EXE} ${RUNDIR}
lgdb
```

When debugging with lgdb you may need to use `continue` to step through after the initial breakpoint.

For further information you can use the interactive help:

```
dbg all> help
```

See `man lgdb` for more information on how to use lgdb.

Comparative Debugging

LGDB contains the Cray comparative debugger… advanced users may which to refer to "Using the Cray Comparative Debugger" document, (S-0042-22) available under http://docs.cray.com

http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-0042-22;idx=books_search;this_sort=title;q=0042;type=books;title=Using%20the%20lgdb%20Comparative%20Debugging%20Feature

OLD-STYLE NAMING AND SYNTAX

Note that in a previous release the STAT commands had different names (STAT, STATview etc.)

Also the –launcher-args option for lgdb used to be –aprun-args