UNDERSTANDING PLACEMENT ON THE CRAY XC30

`acheck` is a C program designed to help users of Cray XC30s understand how Processing Elements (PE) and threads are launched onto the compute nodes and how they are individually bound to CPUs. Note that there are various versions of this document, you should be reading the version appropriate for your system – for example it might have PBSpro. Also note that the Cray documentation mentions a similar program (xthi) that can be used to check binding.

COMPILATION

`acheck` can (and should) be built with all of the compiler tool chains available on the XC30 system, depending on which of the `PrgEnv` modules is loaded. Once the base packages has been unpacked, users should be able to run

```
make
```

To build an executable called:

```
acheck-[cray|intel|gnu]
```

EXECUTION

Also provided is a simple execution script, `run_ECMWF.pbs`, that can be used to submit the job to the scheduler. It has been adapted to use the ECMWF Job Directives and so includes the following lines by default:

```
# ECMWF Job Directives
#PBS -l EC_total_tasks=8
#PBS -l EC_tasks_per_node=8
#PBS -l EC_threads_per_task=1
#PBS -l EC_hyperthreads=1
```

The script then detects which executables have been compiled with each of the programming environments and launches them in turn with the following options:

```
export OMP_NUM_THREADS=$EC_threads_per_task
```

```
aprun -n $EC_total_tasks          \
      -N $EC_tasks_per_node       \
      -d $EC_threads_per_task     \
      -j $EC_hyperthreads         \
      -cc cpu                     \
      ./${exe}
```

For more information on how each of these options affects how the application is launched please refer to the accompanying lecture notes or

```
man aprun
```

UNDERSTANDING THE OUTPUT

By default each test will print a brief summary about how the application was launched. e.g.

```
Testing with acheck-cray
+ aprun -n 8 -N 8 -d 1 -j 1 -cc cpu ./acheck-cray
MPI and OpenMP Affinity Checker v1.0beta

Built with compiler: Cray C++ : Version 8.2.x.x (u82087c82152)

There are 8 MPI Processes on 1 hosts with 8 ranks per process
Each MPI process has 1 OpenMP threads
OMP_NUM_THREADS was 1
```

Each execution should provide the following information:

- The name of the executable being launched
- The aprun launch command being used
- The name and version of the C compiler used to build the executable
- The number of MPI ranks launched
- The number of nodes/hosts used and how many ranks are on each
- The number of threads on each rank

Using this information you may want to attempt the following exercises:

- Check the application is launching the right number of ranks and threads.
  - Is the application using all the available CPUs on the node?
  - If not how can you adjust it to fill the node?
  - What happens when you request more CPUs or nodes than there are nodes?
  - Are the PEs balanced across the CPUs on the numa-nodes? Try using the "`-S`" option to balance the PEs across the numa-nodes.
- Try running the application across more than one node.
  - What changes do you need to make to the PBS run script?
  - What changes (if any) do you need to make to the `aprun` statement?
  - What happens when you get these wrong (e.g. hard coded values to aprun)?
- Try running the application with some OpenMP threads.

- What changes do you have to make to the run script?
- What changes do you have to make the `aprun` command line?
- What if the OMP_NUM_THREADS and the "`-d`" argument do not match? How does this differ to getting MPI ranks and numbers of nodes wrong?

ADVANCED APPLICATION BINDING

Additional information about the binding of individual processors can be shown by adding `-v` as an option to the `acheck` executable, e.g.

```
aprun -n $EC_total_tasks          \
      -N $EC_tasks_per_node       \
      -d $EC_threads_per_task     \
      -j $EC_hyperthreads         \
      -cc cpu                     \
      ./${exe} -v
```

This will produce an additional textual visualisation of which CPUs individuals PEs and their threads are bound to, e.g.

```
                  ---binding---------------------------------------------
    host rank thr  pinning mask
nid00013    0   0   cpu  0 1.......................................
            1   0   cpu  1 .1......................................
            2   0   cpu  2 ..1.....................................
            3   0   cpu  3 ...1....................................
            4   0   cpu  4 ....1...................................
            5   0   cpu  5 .....1..................................
            6   0   cpu  6 ......1.................................
            7   0   cpu  7 .......1................................
```

For each thread all the PE and threads are enumerated and a bitmask showing where each may run is show. A 1 indicates that a CPU may execute on the CPU at that position (from 0-23 with `-j1` and from 0-47 with `-j2`), a dot indicates the PE/thread may not run on that CPU.

Warning running on large numbers of PEs and threads may generate large volumes of data, therefore we recommending limit the exercise to just a few total nodes.

Further Exercises

- Are there any differences between the outputs from different compilers?
  - Look at the output from the Intel Compiler with OpenMP and default binding.
  - What could be causing this (remember the Intel compiler generates an additional helper thread when using OpenMP).
- Try limiting or switching off the binding by passing the `-cc none` or `-cc numa_node` options to `aprun`.
  - What happens to the core affinity? How might this affect application performance?
  - Try using the custom ordering (see `-cc` in `man aprun`) to produce a non-standard binding of threads to CPUs
- Enable the Intel Hyperthreads on the system by adding `-j2` to the `aprun` line?

    - What are the Hyperthread pairs? Which pairs of ranks/threads are being assigned to the same threads?
- Are there any differences between the outputs from different compilers?
    - Look at the output from the Intel Compiler with OpenMP and default binding.
    - What could be causing this (remember the Intel compiler generates an additional helper thread when using OpenMP).
    - Can you think of binding options to fix this?
      Perhaps a custom binding or one of the other binding choices would work.
- What happens if you use MPMD mode?
    - It is legal to use the same binary as the argument to the `aprun` command line. What might this potentially allow you do in your application?

RANK REORDERING

Note that you may not have covered this in the lectures yet, if so just try the first experiment if you have time today and come back to this later on in the week.

Using the `acheck` example provided, investigate how rank reordering affects the placement of tasks on the node.

- If you have not done so already, adjust the default job script so it runs across multiple nodes (i.e. four or more).
- Add "`export MPICH_RANK_REORDER_DISPLAY=1`" to the script.
- Run experiments changing the value of "`export MPICH_RANK_REORDER_METHOD=X`" to 0 (Round-robin), 1(default SMP) and 2 (folded). Note how this changes the hostname each rank ends up on.
- Create a custom rank-reorder file in `MPICH_RANK_ORDER` and add export `MPICH_RANK_REORDER_METHOD=3` to the run script.
    - What happens if this file specifies the wrong number of values?
    - Load the `perftools` module, see `man grid_order` for generating `MPICH_RANK_ORDER` files for Cartesian grid layouts.
- Consider how you might combine using MPMD mode and rank reordering to optimise the behaviour of applications that have individual ranks with different memory or IO requirements (E.g. IO Servers or gather/scatter models).