Batch environment on cca/ccb - PBS

Dominique Lucas
User Support

Some slides courtesy of Cray
Agenda

- **PBS basics**
  - Queues, commands, directives, jobs, …
  - Sequential jobs
  - Tutorial 1

- **Parallel Job submission**
  - ECMWF local PBS directives
  - Fractional and parallel jobs
  - Tutorial 2

- **Advanced job submission**
  - CPU/thread binding
  - Tutorial 3
## Cray XC40 specifications

<table>
<thead>
<tr>
<th>Feature</th>
<th>Phase II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sustained Performance (teraflops)</td>
<td>320</td>
</tr>
<tr>
<td>Peak performance (teraflops)</td>
<td>~8,500</td>
</tr>
<tr>
<td>Processor technology</td>
<td>Intel Broadwell</td>
</tr>
<tr>
<td><strong>Parallel application nodes</strong></td>
<td>3,513</td>
</tr>
<tr>
<td><strong>MOM nodes</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>Pre/Post-processing nodes</strong></td>
<td>104</td>
</tr>
<tr>
<td>Cores per node</td>
<td>36</td>
</tr>
<tr>
<td>Memory per node (GiB)</td>
<td>128 (2400 MHz DDR4)</td>
</tr>
<tr>
<td><strong>External login nodes</strong></td>
<td>2 x Ivy Bridge, 1 x Haswell</td>
</tr>
<tr>
<td>Clock frequency (GHz)</td>
<td>2.1</td>
</tr>
<tr>
<td>Storage capacity (petabytes)</td>
<td>20</td>
</tr>
<tr>
<td>Floating Point Instruction set</td>
<td>AVX2</td>
</tr>
<tr>
<td>Default compiler</td>
<td>Cray 8.4.x</td>
</tr>
</tbody>
</table>
By default, users do not log in and run applications on the compute nodes directly. Instead they launch batch jobs using the following nodes:

- **Pre and Post Processing (PPN)**
  - Used for serial jobs
  - Used for small parallel jobs requiring less than half a node

- **MOM**
  - Where the parallel job scripts run
  - Need to minimise the serial content of such scripts

- **Extreme Scalability Mode (ESM) - Compute Nodes (CN)**
  - Where the full (multi) node parallel executables run
  - Accessed via the command ‘aprun’ (equivalent of mpirun, mpiexec, …) from the MOM node
PBS nodes (commands run on cca)

- pbsnodes –a | less
  
  ccamom05 # this is a MOM node
  
  ntype = PBS
  jobs = 1036567.ccapar/0, 1036565.ccapar/0, 1036364.ccapar/0. …
  resources_available.EC_accept_from_queue = np,dp,tp
  resources_available.vntype = cray_login

  ccappn007 # this is a pre/post processing node (PPN)
  jobs = 1036445.ccapar/1, 1035396.ccapar/2, 1036450.ccapar/3, …
  resources_available.EC_accept_from_queue = os,ts,ns,of,tf,nf,df,ds
  resources_available.vntype = cray_postproc

  cca_2140_0  # this is (half) a Compute node (CN)
  resources_available.EC_accept_from_queue = np,tp,op, dp
  resources_available.vntype = cray_compute

- apstat | less

  Compute node summary
  
  arch config up resv use avail down
  XT  3510  3510  3390  3333  120  0
Batch queues on Crays XC40

<table>
<thead>
<tr>
<th>User Queue Name</th>
<th>Suitable for</th>
<th>Target nodes</th>
<th>Number of processes min/max</th>
<th>Shared / not shared</th>
<th>Processes per node available for user jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns</td>
<td>serial</td>
<td>PPN</td>
<td>1/1</td>
<td>shared</td>
<td>72</td>
</tr>
<tr>
<td>nf</td>
<td>fractional</td>
<td>PPN</td>
<td>2/36</td>
<td>shared</td>
<td>72</td>
</tr>
<tr>
<td>np</td>
<td>parallel</td>
<td>MOM+CN</td>
<td>1/72</td>
<td>not shared</td>
<td>72</td>
</tr>
</tbody>
</table>

- Similar queues for time critical (option 2) work: ts, tf, tp.
- Debug queues are also available: ds, df and dp.
- ‘qstat –Q –f <queue_name> ’ gives full details on specified queue
PBS basics

- Main User commands:

<table>
<thead>
<tr>
<th>User Commands</th>
<th>PBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job submission</td>
<td>qsub [options] &lt;script&gt;</td>
</tr>
<tr>
<td>Job cancel</td>
<td>qdel &lt;job_id&gt;</td>
</tr>
<tr>
<td>Job status</td>
<td>qstat [options] [job_id]</td>
</tr>
<tr>
<td></td>
<td>qscan*</td>
</tr>
<tr>
<td>Queue list</td>
<td>qstat –Q [-f] [queue]</td>
</tr>
<tr>
<td>Check job output</td>
<td>qcat*</td>
</tr>
</tbody>
</table>

(*) ECMWF local commands.

- See man pages for more details.
Requesting resources from PBS

Jobs provide a list of requirements as #PBS comments in the headers of the submission script, e.g.

```
#PBS –l walltime 12:00:00
```

These can be overridden or supplemented at submission by adding to the qsub command line, e.g.

```
> qsub –l walltime 11:59:59 run.pbs
```

Common options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-N &lt;name&gt;</td>
<td>A name for job,</td>
</tr>
<tr>
<td>-q &lt;queue&gt;</td>
<td>Submit job to a specific queues.</td>
</tr>
<tr>
<td>-o &lt;output file&gt;</td>
<td>A file to write the job’s stdout stream in to.</td>
</tr>
<tr>
<td>-e &lt;error file&gt;</td>
<td>A file to write the job’s stderr stream in to.</td>
</tr>
<tr>
<td>-j oe</td>
<td>Join stderr stream in to stdout stream as a single file</td>
</tr>
<tr>
<td>-l walltime <a href="">HH:MM:SS</a></td>
<td>Maximum wall time job will occupy</td>
</tr>
</tbody>
</table>

Jobs must also describe how many compute nodes they will need.
Submitting jobs to the batch system

- Job scripts are submitted to the batch system with qsub:
  - qsub script.pbs
- Once a job is submitted, it is assigned a PBS Job ID, e.g.
  - 1842232.ccapar
- To view the state of all the currently queued and running jobs run:
  - qstat
- To limit to just jobs owned by a specific user
  - qstat –u <username>
- To see details about one job
  - qstat –f <job id>
- To remove a job from the queue, or cancel a running job
  - qdel <job id>
Some queue limits

- Normal queues:
  - User jobs run limit: 20 jobs per queue
  - Time limit: 2 days
  - Memory limit:
    - In queue np: all the memory available (~120GB/node)
    - In queues ns and nf, no memory limits enforced yet. Try to specify the limit you need.

- Time critical queues:
  - Varying user jobs run limits
  - Shorter time limits

- See ‘qstat –Q –f <queue name>’
Shells and accounting.

- ksh and bash shells are supported with PBS
- The ECMWF command ‘eoj’ (end of job) is available and included at the end of each job output file.
  - The resource usage reported in ‘eoj’ is only correct for queues ‘ns’ and ‘nf’. For ‘np’, only resources spent on the MOM node are reported in ‘eoj’. See aprun final report line for the usage of parallel resources.
- Job accounting is running for PBS:
  - All jobs will be charged for elapsed time times the number of cores utilised
  - The accounting currency is the SBU, for System Billing Unit
  - 1 core*hour = 16.11 SBUs
Interactive PBS access – remote X11 in PBS

- ‘Interactive batch’ sessions:

  qsub -I -q np -l EC_nodes=1 -X
  qsub: waiting for job 9848342.ccbpar to start …
  qsub: job 9848342.ccbpar ready
...
  ccbmom06:> aprun …

- From remote (non ECMWF) systems, via ECaccess, to start GUI applications:

  echo $DISPLAY $X11PROTOCOL $X11COOKIE
  136.156.66.24:0.0 MIT-MAGIC-COOKIE-1 3ce0e9a973020f4fe559dd7d108c5195
  qsub -I -q np -l EC_nodes=1 -X
  qsub: waiting for job 9848342.ccbpar to start …
  qsub: job 9848342.ccbpar ready
...
  ccbmom06:> xauth add 136.156.66.24:0.0 MIT-MAGIC-COOKIE-1 3ce0e9a973020f4fe559dd7d108c5195
  ccbmom06:> xclock
PBS at ECMWF - some numbers

- Number of jobs per day per cluster: 250k to 300k
  - Sequential jobs: 50%
  - Fractional jobs: 25%
  - Parallel jobs: 25%

- Average length of jobs: 250 seconds
  - Sequential jobs: 150s
  - Fractional jobs: 85s
  - Parallel jobs: 600s

- Total cost per day per cluster: 45-50M SBUs
  - Sequential jobs: 0.11%
  - Fractional jobs: 0.14%
  - Parallel jobs: 99.74%

- Occupation of // nodes:
  - Operational cluster: 85%
  - Non operational cluster: 98.25%
Pre/Post Processing Mode – queues ns, ds, ts

- Designed to allow multi-user jobs to share compute nodes
  - More efficient for apps running on less than one node
  - Possible interference from other users on the node
- Uses the same fully featured OS as service nodes
- Multiple use cases, applications can be:
  - entirely serial
  - shared memory using OpenMP or other threading model.
  - MPI (limited to intra-node MPI only)
ECMWF job example 1 – sequential job

- cat small_job.cmd
  ...
  #PBS -N Hello_S
  #PBS -q ns
  ...
  cd $SCRATCH
  mars req
  ...
Tutorial 1 – on ccb:

- cd $PERM
- tar xvf ~/trx/pbs.tar
- cd pbs/intro

- Follow instructions in the README file.
- Commands covered:
  - pbsnodes
  - qstat
  - qsub
  - qcat
  - eoj

- See man pages or ‘qcat –h’, ‘eoj –h’ for further help.
Extreme Scaling Mode (ESM) – queues np, dp, tp.

ESM is a high performance mode designed to run larger applications at scale. Important features are:

- Dedicated compute nodes for each user job
  - No interference from other users sharing the node
  - Minimum quanta of compute is 1 node.
- The appropriate parallel runtime environment is automatically set up between nodes

ESM is expected to be the default mode for the majority of applications that require at least one node to run.
To understand how many compute nodes a job needs, we need to understand how parallel jobs are described.

**PE/Processing Element**
- A discrete software process with an individual address space. One PE is equivalent to:
  - 1 MPI Rank or **task**, 1 Co-array Image, 1 UPC Thread, or 1 SHMEM PE

**Threads**
- A logically separated stream of execution inside a parent PE that shares the same address space

**CPU**
- The minimum piece of hardware capable of running a PE. It may share some or all of its hardware resources with other CPUs.
  - Equivalent to a single “Intel Hyperthread”. May also be referred to as ‘Logical CPU’

**Compute Unit**
- The individual unit of hardware for processing, may be seen described as a “core”. May provide one or more CPUs. May also be referred to as ‘Physical CPU’.
Requesting resources from a batch system

- ECMWF Cray XC40s use PBS to schedule resources
  - Users submit batch job scripts to a scheduler from a login node (e.g. PBS) for execution at some point in the future. Each job requests resources and predicts how long it will run.
  - The scheduler (running on an external server) then chooses which jobs to run and when, allocating appropriate resources at the start.
  - The batch system will then execute a copy of the user’s job script on an a one of the “MOM” nodes.
  - The scheduler monitors the job throughout it lifetime. Reclaiming the resources when job scripts finish or are killed for overrunning.

- Each user job script will contain two types of commands
  1. Serial commands that are executed by the MOM node, e.g.
     - quick setup and post processing commands e.g. (rm, cd, mkdir etc)
  2. Parallel launches to run on the allocated compute nodes.
     - Launched using the ‘aprun’ command.
ECMWF have created a bespoke set of job directives for PBS that can be used to define the job core requirements.

The ECMWF job directives provide a direct map between aprun options and PBS jobs.

<table>
<thead>
<tr>
<th>Description</th>
<th>aprun Option</th>
<th>EC Job Directive</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total PEs</td>
<td>-n &lt;n&gt;</td>
<td>#PBS -l EC_total_tasks=&lt;n&gt;</td>
<td>1</td>
</tr>
<tr>
<td>PEs per node</td>
<td>-N &lt;N&gt;</td>
<td>#PBS -l EC_tasks_per_node=&lt;N&gt;</td>
<td>36/72</td>
</tr>
<tr>
<td>Threads per PE</td>
<td>-d &lt;d&gt;</td>
<td>#PBS -l EC_threads_per_task=&lt;d&gt;</td>
<td>1</td>
</tr>
<tr>
<td>CPUs per CU</td>
<td>-j &lt;j&gt;</td>
<td>#PBS -l EC_hyperthreads=&lt;j&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

There are more ‘EC_’ directives.

The PBS prolog will define variables in the job with the same as the names for the ‘EC_’ directives.
Launching ESM Parallel applications

- ALPS : Application Level Placement Scheduler
- aprun is the ALPS application launcher
  - It must be used to run application on the XC compute nodes in ESM mode, (either interactively or as a batch job)
  - If aprun is not used, the application will be run on the MOM node (and will most likely fail).
  - aprun launches sets of PEs on the compute nodes.
  - The aprun man page contains several useful examples
  - The 4 most important parameters to set are:

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number of PEs used by the application</td>
<td>-n</td>
</tr>
<tr>
<td>Number of PEs per compute node</td>
<td>-N</td>
</tr>
<tr>
<td>Number of threads per PE (More precise, the “stride” between 2 PEs on a node)</td>
<td>-d</td>
</tr>
<tr>
<td>Number of to CPUs to use per Compute Unit</td>
<td>-j</td>
</tr>
</tbody>
</table>
ECMWF operational work

- Tight schedule, huge resources needed at short notice
- Three options:
  - Exclusive HPC resources for operational work
  - Sharing of HPC resources with default scheduling of batch work
  - Sharing of resources with enhanced scheduling of batch work
ECMWF’s enhanced scheduling

- Usage of PBS job reservation system.

- For this, we need a precise description of the jobs, including runtime and node requirements.

- The ECMWF PBS filter will set the node requirements and assign an estimated runtime for each job, to guarantee an optimal usage of the resources and a timely delivery of the operational data.

- A jobs runtime database keeps the wall clock time of the last 20 runs for not more than 30 days.

- To benefit from database and scheduling:
  - Keep job name and output file name identical for the same jobs.
Users want something ‘simple’ to write their jobs

The node requirements and job geometry for a job in PBS are specified with a ‘select’ statement (#PBS –l select=..’).

This statement is quite complex to use and doesn’t cover all the requirements needed for the ECMWF enhanced scheduling.

Another complication with PBS is that the user will have to redefine the geometry of his/her run in the script (with ‘aprun’).

At ECMWF, we have therefore decided to customise the PBS environment.

- Users cannot write their own ‘select’ statement.
- ECMWF PBS directives (#PBS –l EC_* ) are available to define the job geometry.
- Environmental variables defining the job geometry are available in the job script.
ECMWF job example 2 – DIY (do it yourself) option

- cat HelloMPIandOpenMP.cmd
  ...
  #PBS -N HelloMPI_OMP
  #PBS -q np
  #PBS -l EC_nodes=3
  ...
  export OMP_NUM_THREADS=6
  aprun -n 36 -N 36 -d 6 -j 2 HelloMPI_OMP
- qsub HelloMPIandOpenMP.cmd
  124950.ccbpar
- qstat -f 124950.ccbpar
  ...
  Resource_List.select=1:vntype=cray_login:EC_accept_from_queue=np:ncpus=0:mem=300MB+3:
  vntype=cray_compute:EC_accept_from_queue=np:mem=120GB

Note the limited memory assigned on the MOM node!
cat HelloMPIandOpenMP.cmd
...
#PBS -N HelloMPI_OMP
#PBS -q np
#PBS -l EC_total_tasks=36
#PBS -l EC_threads_per_task=6
#PBS -l EC_memory_per_task=12GB
#PBS -l EC_hyperthreads=2
...
export OMP_NUM_THREADS=$EC_threads_per_task
aprun -N $EC_tasks_per_node -n $EC_total_tasks \
-d $EC_threads_per_task -j $EC_hyperthreads ./HelloMPI_OMP
...
qstat -f <job_id>
Resource_List.select=select=1:vntype=cray_login:EC_accept_from_queue=np:ncpus=0:mem=3000MB+4:vntype=cray_compute:EC_accept_from_queue=np:mem=115964116992

ECMWF job example 3 – Flexible option
ECMWF job example 4 – MPMD programs, e.g. coupled runs

```
> cat MPMD.cmd

.. 
#PBS -N HelloMPMD
#PBS -q np
#PBS -l EC_total_tasks=30:6
#PBS -l EC_threads_per_task=1:1
#PBS -l EC_hyperthreads=1
...
IFS=':'
tasks_per_node=($EC_tasks_per_node)
total_tasks=($EC_total_tasks)
aprun -n ${total_tasks[0]} -N ${tasks_per_node[0]} ./model_atm :
    -n ${total_tasks[1]} -N ${tasks_per_node[1]} ./model_ocean
...```

With this method, one cannot run two different executables on the same node, therefore we may be wasting compute resources.
ECMWF job example 5 – MPMD programs, improved recipe

- cat MPMD_turbo.cmd
  
  ```
  #PBS -N HelloMPMD
  #PBS -q np
  #PBS -l EC_nodes=1

  cat>wrapper.sh<<eof
  #!/bin/ksh
  rank=$EC_FARM_ID
  export OMP_NUM_THREADS=1
  if [ $rank -lt 30 ]; then
    exec ./atm>atm.out.$rank 2>&1
  else
    exec ./ocean>ocean.out.$rank 2>&1
  fi
  eof
  ```

  export PMI_NO_FORK=1
  # this variable is essential
  aprun --n36 --N36 -j1 ./wrapper.sh

  # $EC_FARM_ID is defined for # you for each entity of the # wrapper.sh script running.


No waste of CPU resources!
Running applications on the Cray XC40: Some basic examples

Assuming XC40 nodes with 2x18 core Ivybridge processors

- Each node has: 72 CPUs/Hyperthreads and 36 Compute Units/cores

• Launching an MPI application on all CPUs of 64 nodes:
  - Using 1 CPU per Compute Unit means a maximum of 36 PEs per node.
  - 64 nodes x 36 ranks/node = 2304 ranks
    $ aprun –n 2304 –N 36 –j 1 ./model-mpi.exe

• Launch the same MPI application on 64 nodes but with half as many ranks per node
  - Still using 1 CPU per Compute Unit, but limiting to 18 Compute Units.
    $ aprun –n 1152 –N 18 –j 1 ./model-mpi.exe
  - Doubles the available memory for each PE on the node

• To use all available CPUs on 64 nodes.
  - Using 2 CPUs per Compute unit, so 72 PEs per node
    $ aprun –n 4608 –N 72 –j 2 ./model-mpi.exe
Some examples of hybrid invocation

- To launch a Hybrid MPI/OpenMP application on 64 nodes
  - 384 total ranks, using 1 CPU per Compute Unit (Max 36 Threads)
  - Use 6 PEs per node and 6 Threads per PE
  - Threads set by exporting OMP_NUM_THREADS
    
    ```
    $ export OMP_NUM_THREADS=6
    $ aprun -n 384 -N 6 -d $OMP_NUM_THREADS -j 1 ./model-hybrid.exe
    ```

- Launch the same hybrid application with 2 CPUs per CU
  - Still 384 total ranks, using 2 CPUs per Compute Unit (Max 72 Threads)
  - Use 6 PEs per node and 12 Threads per PE
    
    ```
    $ export OMP_NUM_THREADS=12
    $ aprun -n 384 -N 6 -d $OMP_NUM_THREADS -j 2 ./model-hybrid.exe
    ```
Watching a launched job on the Cray XE

- **xtnodestat**
  - Shows how ESM nodes are allocated and corresponding aprun commands

- **apstat**
  - Shows aprun processes status
    - *apstat* overview
    - *apstat* –a [ apid ] info about all the applications or a specific one
    - *apstat* –n info about the status of all the nodes
    - *apstat* –a –v [ apid ] more detailed info about all applications (or a specific one), including PBS job ID.

- **qstat**
  - shows batch jobs or queues
Pre/Post Processing Mode – queues nf, df, tf

- Designed to allow multi-user jobs to share compute nodes
  - More efficient for apps running on less than one node
  - Possible interference from other users on the node
- Uses the same fully featured OS as service nodes
- Multiple use cases, applications can be:
  - entirely serial
  - shared memory using OpenMP or other threading model.
  - MPI (limited to intra-node MPI)
ECMWF job example 6 – fractional job

```bash
# cat small_job.cmd
...
#PBS -N HelloMPI_S
#PBS -q nf
#PBS -l EC_total_tasks=18
#PBS -l EC_hyperthreads=2
...
module load cray-snplauncher
mpiexec -n $EC_total_tasks ./HelloMPI_S
...
```
How to manage dependencies between jobs

- Typical 3-steps job, with dependencies:
  - fetch data - serial
  - model run - parallel
  - post processing - serial

- Alternatives are to:
  - DIY: submit subsequent job at the end of current job.
  - use the ‘qsub –W depend’ option
  - use ‘qsub –W block=true’ option
  - use a scheduling system, e.g. ECMWF’s ecFlow software.
Tutorial 2 – on ccb:

➢ cd $PERM/pbs/parallel

● Follow instructions in the README file. You will submit fractional and parallel jobs.

● Commands covered:
  - xtnodestat
  - apstat
  - mpiexec
  - aprun
  - qsub, qstat, maybe qdel …

● See relevant man pages. For the ECMWF PBS EC_* directives, see under https://software.ecmwf.int/wiki/display/UDOC/EC_+Job+Directives+Summary
Default Binding - CPU

- By default aprun will bind each PE (MPI task or rank) to a single CPU for the duration of the run.
- This prevents PEs moving between CPUs.
- All child processes of the PE are bound to the same CPU
- PEs are assigned to CPUs on the node in increasing order from 0. e.g.

```
0 0
1 1
2 2
... 35 35
```

Node 0

1 Software PE
is bound to 1
Hardware CPU

```
36 0
25 1
26 2
... 71 35
```

Node 1

aprun -n 72 -N 36 -j1 a.out
You can inform aprun how many threads will be created by each PE by passing arguments to the –d (depth) flag.

aprun does not create threads, just the master PE.

PEs are bound to CPU spaced by the depth argument, e.g.

```
Node 0

0 1 2 3

Node 1

18 19 35
```

1 Software PE is bound to 1 Hardware CPU

```
aprun -n 36 -N 18 -d2 -j1 a.out
```
Default Thread Binding (pt 2)

- Each subsequently created child processes/thread is bound by the OS to the next CPU (modulo by the depth argument).
  e.g.

```bash
OMP_NUM_THREADS=2
aprun -n 36 -N 18 -d2 -j1 a.out
```

- Each PE becomes the master thread and spawns a new child thread. The OS binds this child thread to the next CPU.
Default Thread Binding (pt 3)

- aprun cannot prevent PEs from spawning more threads than requested
- In such cases threads will start to “wrap around” and be assigned to earlier CPUs.
- E.g.:

```
OMP_NUM_THREADS=3
aprun -n 36 -N 18 -d2 -j1 a.out
```

In this case, the third thread is assigned to the same CPU as the master PE causing contention for resources.
Removing binding entirely

- aprun can be prevented from binding PEs and their children to CPUs, by specifying “–cc none”. E.g.:

  OMP_NUM_THREADS=3
  aprun -n 36 -N 18 --cc none -j1 a.out

- All PEs and their child processes and threads are allowed to migrate across cores as determined by the standard Linux process scheduler.
- This is useful where PEs spawn many short lived children (e.g. compilation scripts) or over-subscribe the node.
- (-d removed as it no longer serves a purpose)
The design of the XC node means that CPUs accessing data stored on the other socket/die have to cross the QPI inter-processor bus.

This is marginally slower than accessing local memory and creates “Non-Uniform Memory Access” (NUMA) regions.

Each XC node is divided into two NUMA nodes, associated with the two sockets/dies.
NUMA nodes and CPU binding (pt 1)

- Care has to be taken when under-populating node (running fewer PEs than available CPUs). E.g.

```
aprun -n 36 -N 18 -j1 a.out
```

- The default binding will bind all PEs to CPUs in the first NUMA node of each node.
- This will unnecessarily push all memory traffic through only one die’s memory controller. Artificially limiting memory bandwidth.
NUMA nodes and CPU binding (pt 2)

- The -S <PEs> flag tells aprun to distribute that many PEs to each NUMA node, thus evening the load.

```
aprun -n 36 -N 18 -S 9 -j1 a.out
```

- PEs will be assigned to CPUs in the NUMA node in the standard order, e.g. 0-8, 9-17, 18-26 and 27-35. However all CPUs within a NUMA node are essentially identical so there are no additional imbalance problems.
As well as completely removing binding, it is also possible to make aprun bind PEs to all the CPUs on a NUMA node.

```
aprun -n 36 -N 18 -S 9 -j1 --cc numa_node a.out
```

PEs will be assigned to the NUMA node that their original PE would have been assigned to with CPU binding and the same options.

OS allowed to migrate processes within the NUMA node, should be better performance than no binding. “–cc none”
Binding to a CPU set: -depth

- An extension to “numa_node” is the option -cc depth.
- depth defines that a ‘cpu set’ should be used where all threads belonging to a rank are “unbound”. The size of the cpu set is given by the –d option.

```
 aprun -n 6 -d4 --cc depth -j1 a.out
```

- Solves the ‘Intel Helper Thread’ issue and also the ‘oversubscribing’ of threads.
Each XC node is an shared memory device.

By default all memory is placed on the NUMA node of the first CPU to “touch” it.

However, it may be beneficial to setup strict memory containment between NUMA nodes.

This prevents PEs from one NUMA node allocating memory on another NUMA node.

This has been shown to improve performance in some applications.

```bash
aprun -ss -n 72 -N 18 -S 9 a.out
```
Ignore Hyperthreads; “-j1” Single Stream Mode

All examples up to now have assumed “-j1” or “Single Stream Mode”.

In this mode, aprun binds PEs and ranks to the 36 Compute Units (e.g. only use CPUs 0-35)
Include Hyperthreads “-j2” Dual Stream Mode

Specifying “-j2” in aprun assigns PEs to all of the 72 CPUs available. However CPUs that share a common Compute Unit are assigned consecutively.

This means threads will share Compute Units with default binding.
Custom Binding

- aprun also allows users to customise the binding of PEs to CPUs.
  - Users may pass a colon separated list of CPU binding options to the –cc option.
  - The $n^{th}$ PE on the node is bound by the $n^{th}$ binding option.

- Each PE binding option may be either a single CPU or a comma separated list of CPUs.
  - Specifying a single CPU forces the PE and all children and threads to the same PE
  - Specifying a comma separated list binds the PE to the first CPU in the list and children and threads on to the
    subsequent CPUs (round-robin)
  - Additional PEs will be left unbound.
Custom Binding (example)

- Custom binding can be hard to get right. The xthi application is useful for testing binding.
  - Source code available in S-2496 (Workload Management and Application Placement for the Cray Linux Environment) Section 8.7 at docs.cray.com (http://docs.cray.com/books/S-2496-5202/S-2496-5202.pdf)

```bash
> export OMP_NUM_THREADS=2
> aprun -n 4 -N 4 --cc 3,2:7,8:9,10,4:1 xthi | sort
Hello from rank 0, thread 0, on nid00009. (core affinity = 3)
Hello from rank 0, thread 1, on nid00009. (core affinity = 2)
Hello from rank 1, thread 0, on nid00009. (core affinity = 7)
Hello from rank 1, thread 1, on nid00009. (core affinity = 8)
Hello from rank 2, thread 0, on nid00009. (core affinity = 9)
Hello from rank 2, thread 1, on nid00009. (core affinity = 10)
Hello from rank 3, thread 0, on nid00009. (core affinity = 1)
Hello from rank 3, thread 1, on nid00009. (core affinity = 1)
```
Follow instructions in the README file. You will look at different ways of binding a parallel application to the hardware.

Commands covered:
- aprun
- qsub, qstat, maybe qdel …

See ‘aprun’ man page.