Performance tips

In order to help you to get the best performance with your compute as possible, thereafter a few tips/advises I can give:

For single node jobs only

IO tunings

For single/mono node jobs only, if you are using your workdir for your own compiled stuffs, i advise you to tar the top dir of and untar it into /scratch/<job-ID> when you are running your jobs. Indeed, node disk IO will better perform due to no IO competition on.

For example, based on the premise you have an **appdir** directory in your own environment, each time you compile/install something in, you just have to [re]create/update the tar file doing on master:

```
$ tar rf $(/shared/scripts/getWorkdir.sh)/packages.tar appdir
```

Now, when you are running a single node job, you can add these following lines into your job script in the USER-part section:

```
[...]
# tar xf /workdir/<groupname>/<username>/packages.tar -C $PWD
export ROOT_WORKDIR=/$(/shared/scripts/getWorkdir.sh|awk -F'/' '{print
$2"/"$3"/"$4}')
tar xf $ROOT_WORKDIR/packages.tar -C $WORKDIR/

export LD_LIBRARY_PATH=$WORKDIR/appdir/lib:$LD_LIBRARY_PATH
export PATH=$WORKDIR/appdir/bin:$LD_LIBRARY_PATH

mpirun <what you want>
[...]
```

MPI tunings

When you are using OpenMPI for launching single node jobs, you should better to avoid the network usage for communication. Indeed, internal communications will still perform faster than network ones.

To do so, in interactive sessions or into your job scripts, you can either:

- set several environment variables:

```
$ export PSM_DEVICES=self,shm
$ export OMPI_MCA_mtl=^psm
$ export OMPI_MCA_btl=shm,self
$ mpirun ...
```

- add in **mpirun** parameters the expected options:

```
$ mpirun -mca mtl ^psm -mca btl shm,self -x PSM_DEVICES=self,shm ...
```

More general run-time tunings

In general, a good way to get the best application performance is to well tune the processor and memory affinity.

To do so, here 2 ways to proceed:

- with OpenMPI, you can provide in parameters to **orterun**, **mpirun**, **mpiexec** these following options:
  - **–bind-to-core**: bind processes to cores (my favorite one)
  - **–bind-to-socket**: bind processes to processor sockets
  - **–bind-to-none**: not not bind processes

```
$ mpirun --bind-to-core simulation.x
```

- without any parallel launcher, you can bind by yourself your processes by using one of the following commandes:
  - **taskset**
  - **numactl**
  - **pin_t2c**

taskset

```
$ taskset -c 0,1,2,3 simulation.x
```

numactl

```
$ numactl --physcpubind=0,1,2,3 simulation.x
```

Here you run your **simulation.x** application on the 4 first cores of the first socket processor.

Because we are using **CGROUP/CPUSET** Linux kernel features to contain jobs under dedicated sets of cores, before trying to use **taskset** or **numactl** command, you should get your core IDs provided by Torque doing:
$ cat /dev/cpuset/torque/$PBS_JOBID/cpus

pin_t2c

You can also use a homemade tool named **pin_t2c** (from eponymous module) which provides an easy way to pin threads of a running process to hardware resources

```
$ module load pin_t2c
$ myapp & pin_t2c --pid $!
```

** without any other optional argument, this take care of the complete CGROUP/CPUSET allocation provided by Torque; but you can manually specify CPU sockets, cores, etc. to well tune your process.*

Embarrassingly parallel jobs

For those who need to run embarrassingly parallel jobs (non-MPI processes), you have 2 options for:

- with **OpenMPI** and **mpirun**/**mpiexec** binary:

Example, inside the User-PART section of your submission script:

```
[...]
module load openmpi
export PSM_DEVICES=self,shm
export OMPI_MCA_mtl=^psm
export OMPI_MCA_btl=shm,self

mpirun --bind-to core simulation.x
[...]
```

If the task is identical on all processor cores, you can simply use it without any flourish; but in the case where each processor core should be doing a different task, then you can make use of an environmental variable called **$OMPI_COMM_WORLD_RANK** (a kind of global core IDs) which will be different for every running task.

- with **Torque** bundled utilities: you can use the **pbsdsh** tool contained in torque client module.

Example, inside the User-PART section of your submission script:

```
[...]
module load torque
pbsdsh -v $WORKDIR/taskscript.sh
[...]
```

taskscript.sh

```
#!/bin/bash -l

source /usr/share/Modules/init/bash 2> /dev/null # to be able to load some modules

module load <what-you-need>
simulation.x
```

If the task is identical on all processor cores, you can simply use it without any flourish; but in the

case where each processor core should be doing a different task, then you can make use of an environmental variable called **$PBS_VNODENUM** (a kind of global core IDs) which will be different for every running task.

You have to keep in mind a script invoked by pbsdsh starts in a very basic environment and dont inherit your own environment. So, dont forget to reconstruct your needed shell environment in the task script.