CPU affinity

CPU affinity is the name for the mechanism by which a process is bound to a specific CPU (core) or a set of cores.

For some background here’s an article from 2003 when this capability was introduced to Linux for the first time: http://www.linuxjournal.com/article/6799

Another good overview is provided by Glenn Lockwood of the San Diego Supercomputer Center at http://www.glennlockwood.com/hpc-howtos/process-affinity.html

- NUMA
- What’s the problem?
  - How do I use CPU affinity?
- Note on Masks
- SLURM and srun
  - MVAPICH2
  - Intel MPI
- OpenMP CPU affinity
  - Intel
  - GNU
- Things you probably don’t want to know about:
  - “FEATURE” ALERT FOR Intel Infiniband (Bellatrix and Deneb)
  - CGroups
- Related articles

NUMA

Or as it should be, ccNUMA, *cache coherent non uniform memory architecture*. All modern multi socket computers look something like the diagram below with multiple levels of memory and some of this being distributed across the system.

Memory is allocated by the operating system when asked to do so by your code but the physical location is not defined until the moment at which the memory page is accessed. The default is to place the page on the closest physical memory (i.e. the memory directly attached to the socket) as this provides the highest performance. If the thread accessing the memory moves to the other socket the memory will not follow!

Cache coherence is the name for the process that ensures that if one core updates information that is also in the cache of another core the change is propagated.

What’s the problem?

Apart from the memory access already discussed, if we have exclusive nodes with only one mpirun per node then there isn’t a problem as everything will work “as designed”. The problems begin when we have shared nodes, that is to say nodes with more than one mpirun per system image. These mpiruns may all belong to the same user. In this case the default settings can result in some very strange and unwanted behaviour.

If we start mixing flavours of MPI on nodes then things get really fun....

Hybrid codes, that is to say mixing MPI with threads or OpenMP also present a challenge. By default Linux threads inherit the mask of the spawning process so if you want your threads to have free use of all the available cores please take care!
How do I use CPU affinity?

The truthful and unhelpful answer is:

```c
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sched.h>
int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

What is more usual is that something else (i.e. the MPI library and mpirun) sets it for your processes as it sees fit or as you ask it by setting some variables. As we will see the behaviour is somewhat different between MPI flavours!

It's also possible to use the taskset command line utility to set the mask

```bash
:- > taskset 0x00000003 mycommand
```

Note on Masks

When talking about affinity we use the term "mask" or "bit mask" which is a convenient way of representing which cores are part of a CPU set. If we have an 8 core system then the following mask means that the process is bound to CPUs 7 & 8.

```
11000000
```

This number can be conveniently written in hexadecimal as c0 (192 in decimal) and so if we query the system regarding CPU masks we will see something like:

```
pid 8092's current affinity mask: 1c0
pid 8097's current affinity mask: 1c0000
```

In binary this would translate to

```
pid 8092's current affinity mask: 000111000000
pid 8097's current affinity mask: 000111000000000000000000
```

This shows that the OS scheduler has the choice of three cores on which it can run these single threads.

SLURM and srun

As well as the traditional MPI process launchers (mpirun) there is also srun which is SLURM's native job starter. Its main advantages are its tight integration with the batch system and speed at starting large jobs.

In order to set and view CPU affinity with srun one needs to pass the "--cpu_bind" flag with some options. We strongly suggest that you always ask for "verbose" which will print out the affinity mask.

To bind by rank:
<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
</table>
| `srun -N 1 -n 4 -c 1 --cpu_bind=verbose,rank ./hi 1` | CPU binding by rank: 
- CPU 0: `mask 0x1 set`
- CPU 1: `mask 0x2 set`
- CPU 3: `mask 0x8 set`
- CPU 2: `mask 0x4 set`
- Hello world, b370
0: `sleep(1)`
0: `bye-bye`
1: `sleep(1)`
1: `bye-bye`
2: `sleep(1)`
2: `bye-bye`
3: `sleep(1)`
3: `bye-bye` |
| `srun -N 1 -n 4 -c 4 --cpu_bind=verbose,sockets ./hi 1` | CPU binding to sockets: 
- CPU 1: `mask 0xff set`
- CPU 0: `mask 0xff set`
- CPU 2: `mask 0xff set`
- CPU 3: `mask 0xff00 set`
- Hello world, b370
0: `sleep(1)`
0: `bye-bye`
2: `sleep(1)`
2: `bye-bye`
1: `sleep(1)`
1: `bye-bye`
3: `sleep(1)`
3: `bye-bye` |

Please be aware that binding by rank is only recommended for pure MPI codes as any OpenMP or threaded part will also be confined to one CPU!

To bind to sockets:

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
</table>
| `srun -N 1 -n 4 -c 4 --cpu_bind=verbose,sockets ./hi 1` | CPU binding to sockets: 
- CPU 1: `mask 0xff set`
- CPU 0: `mask 0xff set`
- CPU 2: `mask 0xff set`
- CPU 3: `mask 0xff00 set`
- Hello world, b370
0: `sleep(1)`
0: `bye-bye`
2: `sleep(1)`
2: `bye-bye`
1: `sleep(1)`
1: `bye-bye`
3: `sleep(1)`
3: `bye-bye` |

To bind with whatever mask you feel like:
In the case of there being an exact match between the number of tasks and the number of cores srun will bind by rank but by default there is no CPU binding:

```
~> srun -N 1 -n 4 -c 4 --cpu_bind=verbose,mask_cpu:f,0,000 ./hi 1

cpu_bind=MASK - b370, task  0  0 [5408]: mask 0xf set
cpu_bind=MASK - b370, task  1  1 [5409]: mask 0xf0 set
cpu_bind=MASK - b370, task  2  2 [5410]: mask 0xf00 set
cpu_bind=MASK - b370, task  3  3 [5411]: mask 0xf000 set

Hello world, b370
0: sleep(1)
0: bye-bye

Hello world, b370
1: sleep(1)
1: bye-bye

Hello world, b370
3: sleep(1)
3: bye-bye

Hello world, b370
2: sleep(1)
2: bye-bye
```

This may well result in sub optimal performance as one has to rely on the OS scheduler to (not) move things around.

See the `--cpu_bind` section of the `srun` man page for all the details!

**MVAPICH2**

MVAPICH2-CH3 interfaces support architecture specific CPU mapping through the Portable Hardware Locality (hwloc) software package. By default, the HWLOC sources are compiled and built while the MVAPICH2 library is being installed. Users can choose the `--disable-hwloc` parameter while configuring the library if they do not wish to have the HWLOC library installed. However, in such cases, the MVAPICH2 library will not be able to perform any affinity related operations.

There are two placement options (bunch and scatter) and one needs to explicitly turn off CPU affinity with `MV2_ENABLE_AFFINITY=0` if it's not wanted.

The default behaviour is to place processes by rank so that rank 0 is on core 0 and rank 1 is on core 1 and so on.

This means that if there are two (or more) MAVPICH2 MPI jobs on the same node they will both pin their processes to the same cores! Therefore two 8 way MPI jobs on a 16 core node will use only the first 8 cores and will both run at 50% thanks to CPU timesharing.

The best case seen so far involved 48 rank 0 processes sharing the first core on a 48 way node!

If you need to run multiple instances of a code using MVAPICH2 on the same node then it's vital to tell SLURM to use different CPUs for each task!
Here the --exclusive flag has a different meaning than when used as a #SBATCH directive. As usual the man page explains this in great detail:

```bash
$ man srun
```

--exclusive[=user|mcs]

This option applies to job and job step allocations, and has two slightly different meanings for each one. When used to initiate a job, the job allocation cannot share nodes with other running jobs (or just other users with the "=user" option or "=mcs" option).

The default shared/exclusive behavior depends on system configuration and the partition's OverSubscribe option takes precedence over the job's option.

This option can also be used when initiating more than one job step within an existing resource allocation, where you want separate processors to be dedicated to each job step. If sufficient processors are not available to initiate the job step, it will be deferred. This can be thought of as providing a mechanism for resource management to the job within it's allocation.

The exclusive allocation of CPUs only applies to job steps explicitly invoked with the --exclusive option. For example, a job might be allocated one node with four CPUs and a remote shell invoked on the allocated node. If that shell is not invoked with the --exclusive option, then it may create a job step with four tasks using the --exclusive option and not conflict with the remote shell's resource allocation.

Use the --exclusive option to invoke every job step to insure distinct resources for each step.

Note that all CPUs allocated to a job are available to each job step unless the --exclusive option is used plus task affinity is configured. Since resource management is provided by processor, the --ntasks option must be specified, but the following options should NOT be specified:

```
--relative, --distribution=arbitrary. See EXAMPLE below.
```

**Intel MPI**

By default Intel MPI is configured to use srun but it's possible to use the "native" mpirun.

If you do this it's important to tell it not to use the SLURM PMI and to disable CPU binding within SLURM:

```
$ unset I_MPI_PMI_LIBRARY
$ export SLURM_CPU_BIND=none
```

Once these variables have been unset/set it is possible to launch tasks with mpirun. The main environmental variables are:

`I_MPI_PIN` - Turn on/off process pinning. Enable process pinning. The default is that it is activated.

`I_MPI_PIN_MODE` - Choose the pinning method. Pin processes inside the process manager involved (Multipurpose Daemon*/MPD or Hydra*). This is the default value.
Then for the `mpirun.hydra`

- `I_MPI_PIN_RESPECT_CPUSERT` - Respect the process affinity mask. Respect the process affinity mask. This is the default value
- `I_MPI_PIN_RESPECT_HCA` - In the presence of Infiniband architecture* host channel adapter (IBA* HCA), adjust the pinning according to the location of IBA HCA. Use the location of IBA HCA (if available). This is the default value

The behaviour by default is to share the node between processes so a two way job on a 16 core nodes results in two process with the following masks:

<table>
<thead>
<tr>
<th>ff</th>
<th>0000000011111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00ff</td>
<td>1111111100000000</td>
</tr>
</tbody>
</table>

Likewise a 16 way process on a 48 core node gives masks of the form:

| 000000000000000000000000000000000000000000000111 |
| 000000000000000000000000000000000000000000111000 |

and so on... This makes sense for hybrid jobs but would be less than optimal for situations where one wants to run a pure MPI code with fewer ranks than processors.

### OpenMP CPU affinity

There are two main ways that OpenMP is used on the clusters.

1. A single node OpenMP code
2. A hybrid code with one OpenMP domain per rank

For both Intel and GNU OpenMP there are environmental variables which control how OpenMP threads are bound to cores.

The first step for both is to set the number of OpenMP threads per job (case 1) or MPI rank (case 2). Here we set it to 8

```
export OMP_NUM_THREADS=8
```

**Intel**

The variable here is `KMP_AFFINITY`

- `export KMP_AFFINITY=verbose,scatter`  # place the threads as far apart as possible
- `export KMP_AFFINITY=verbose,compact`  # pack the threads as close as possible to each other

The official documentation can be found at [https://software.intel.com/en-us/node/522691](https://software.intel.com/en-us/node/522691)

**GNU**

with GCC one needs to set either

**OMP_PROC_BIND**

- `export OMP_PROC_BIND=SPREAD`  # place the threads as far apart as possible
- `export OMP_PROC_BIND=CLOSE`  # pack the threads as close as possible to each other

or `GOMP_CPU_AFFINITY` which takes a list of CPUs
GOMP_CPU_AFFINITY="0 2 4 6 8 10 12 14"   # place the threads on CPUs 0,2,4,6,8,10,12,14 in this order.
GOMP_CPU_AFFINITY="0 8 2 10 4 12 6 14"   # place the threads on CPUs 0,8,2,10,4,12,6,14 in this order.

The official documentation can be found at https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html#Environment-Variables

Things you probably don’t want to know about:

"FEATURE" ALERT FOR Intel Infiniband (Bellatrix and Deneb)

Either disabling affinity (IntelMPI with I_MPI_PIN=0) or having no affinity but leaving I_MPI_FABRICS=shm:tmi (required for Qlogic Infiniband) as set by the module results in very strange behaviour! For example on a 16 core node we see

```
[me@mysystem hello]$ mpirun -genv I_MPI_PIN 0 -n 2 -hosts b001 -genv I_MPI_FABRICS=shm:tmi `pwd`/hello 60
root@b001:~ > taskset -p 17032
pid 17032's current affinity mask: 1
root@b001:~ > taskset -p 17033
pid 17033's current affinity mask: 2
```

Instead of the expected masks of fff and fff we have 10 and 01. This is caused by the driver for the QLogic Infiniband cards and in order to fully disable pinning one needs to set the following variable

```
IPATH_NO_CPUAFFINITY=1
```

Section 4-22 of the QLogic(Intel OFED+ software guide explains that:

InfiniPath attempts to run each node program with CPU affinity set to a separate logical processor, up to the number of available logical processors. If CPU affinity is already set (with sched_setaffinity() or with the taskset utility), then InfiniPath will not change the setting ..... To turn off CPU affinity, set the environment variable IPATH_NO_CPUAFFINITY

Caveat emptor and all that...

On the SCITAS clusters we add this setting to the module so when the MPI launcher doesn’t set affinity there is also nothing set by the QLogic driver.

CGroups

As CGroups and tasksets both do more or less the same thing it’s hardly surprising that they aren’t very complementary.

The basic outcome is that if the restrictions imposed aren’t compatible then there’s an error and the executable isn’t run. Even if the restrictions imposed are compatible they may still give unexpected results.

One can even have unexpected behaviour with just CGroups! A nice example of this is creating an 8 core CGroup and then using IntelMPI with pinning activated to run mpirun -np 12 ./mycode . The first eight processes have the following masks

```
10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
```

The next four then have

```
10000000
01000000
00100000
00010000
```
So eight processes will timeshare and four will have full use of a core. If pinning is disabled then all processes have the mask ff so will timeshare.

Related articles

- Connecting to the clusters
- Running Docker images using Shifter
- Using ANSYS with Remote Solve Manager
- Licensed Software on the Clusters
- Using the clusters