Managing Process Affinity in Linux

Home (http://www.glennklockwood.com/)
: HPC Howtos (http://www.glennklockwood.com/hpc-howtos/) : Process Affinity

Contents

- 1. Introduction
- 2. Types of Thread Scheduling
  - 2.1. Compact Scheduling
  - 2.2. Round-Robin Scheduling
  - 2.3. Stupid Scheduling
- 3. Defining Affinity
  - 3.1. The Linux-Portable Way (taskset)
  - 3.2. The Other Linux-Portable Way (numactl)
  - 3.3. Using OpenMP Runtime Extensions
  - 3.4. Using OpenMP 4.0 Runtime Controls
  - 3.5. getfreesocket

1. Introduction

Although a compute node or workstation may appear to have 16 cores and 64 GB of DRAM, these resources are not uniformly accessible to your applications. The best application performance is usually obtained by keeping your code’s parallel workers (e.g., threads or MPI processes) as close to the memory on which they are operating as possible. While you might like to think that the Linux thread scheduler would do this automatically for you, the reality is that most HPC applications benefit greatly from a little bit of help in manually placing threads on different processor cores.

To get an idea of what your multithreaded application is doing while it is running, you can use the `ps` command.

Assuming your executable is called `application.x`, you can easily see what cores each thread is using by issuing the following command in bash:

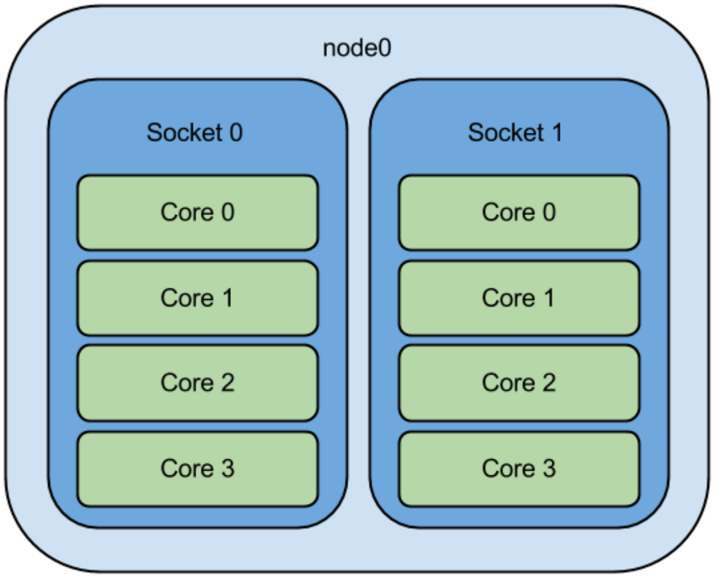
```
$  for i in $(pgrep application.x); do ps -mo pid,tid,fname,user,psr -p $i;done
```

The PSR field is the OS identifier for the core each TID (thread id) is utilizing.

2. Types of Thread Scheduling

Certain types of unevenly loaded applications can experience serious performance degradation caused by the Linux scheduler treating high-performance application codes in the same way it would treat a system daemon that might spend most of its time idle.

These sorts of scheduling issues are best described with diagrams. Let’s assume we have compute nodes with two processor sockets, and each processor has four cores:

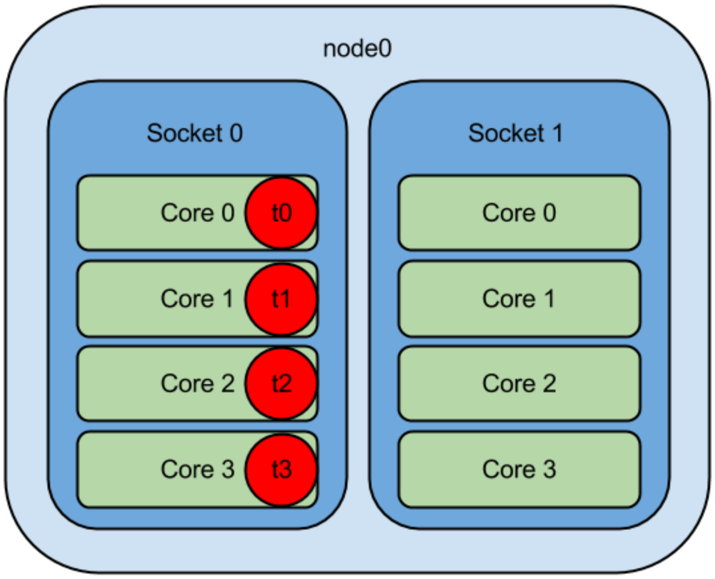


(scheduling-topology.png)

When you run a multithreaded application with four threads (or even four serial applications), Linux will *schedule* those threads for execution by assigning each one to a CPU core. Without being explicitly told how to do this scheduling, Linux may decide to

1. run thread0 to thread3 on core0 to core3 on socket0
2. run thread0 and thread1 on core0 and core1 on socket0, and run thread2 and thread3 on socket1
3. run thread0 and thread1 on core0 only, run thread2 on core1, run thread3 on core2, and leave core3 completely unutilized
4. any number of other nonsensical allocations involving assigning multiple threads to a single core while other cores sit idle

It should be obvious that option #3 and #4 are very bad for performance, but the fact is that Linux will happily schedule your multithreaded job (or multiple single-thread jobs) this way if your threads behave in a way that is confusing to the operating system.

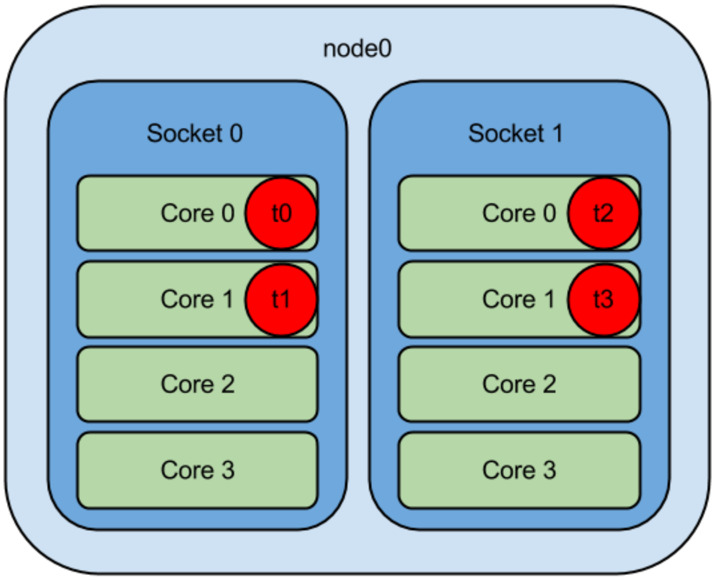


(scheduling-compact.png)

2.1. Compact Scheduling

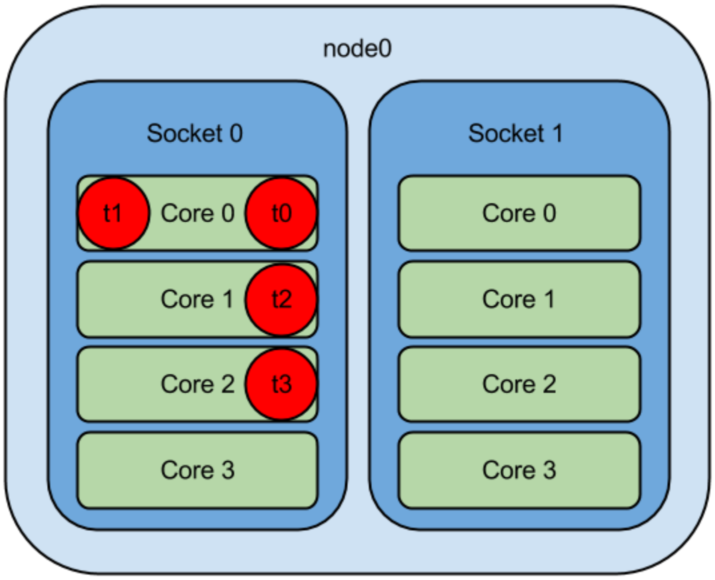
Option #1 is often referred to as “compact” scheduling and is depicted in the diagram to the right. It keeps all of your threads running on a single physical processor if possible, and this is what you would want if all of the threads in your application need to repeatedly access different parts of a large array. This is because all of the cores on the same physical processor can access the memory banks associated with (or “owned by”) that processor at the same speed. However, cores cannot access memory stored on memory banks owned by a different processor as quickly; this is phenomenon is called NUMA (non-uniform memory access). If your threads all need to access data stored in the memory owned by one processor, it is often best to put all of your threads on the processor who owns that memory.

2.2. Round-Robin Scheduling



Option #2 is called “scatter” or “round-robin” scheduling and is ideal if your threads are largely independent of each other and don’t need to access a lot of memory that other threads need. The benefit to round-robin thread scheduling is that not all threads have to share the same memory channel and cache, effectively doubling the memory bandwidth and cache sizes available to your application. The tradeoff is that memory latency becomes higher as threads have to start accessing memory that might be owned by another processor.

2.3. Stupid Scheduling



(scheduling-stupid.png)

Option #3 and #4 are what I call "stupid" scheduling (see diagram to the right) and can often be the default behavior of the Linux thread scheduler if you don't tell Linux where your threads should run. This happens because in traditional Linux server environments, most of the proceses that are running at any given time aren't doing anything. To conserve power, Linux will put a lot of these quiet processes on the same processor or cores, then move them to their own dedicated core when they wake up and have to start processing.

If your application is running at full bore 100% of the time, Linux will probably keep it on its own dedicated CPU core. However, if your application has an uneven load (e.g., threads are mostly idle while the last thread finishes), Linux will see that the application is mostly quiet and pack all the quiet threads (e.g., t0 and t1 in the diagram to the right) on to the same CPU core. This wouldn't be so bad, but the cost of moving a thread from one core to another requires *context switches* which get very expensive when done hundreds or thousands of times a minute.

3. Defining affinity

There are several ways to specify how you want your threads to be bound to cores.

- **If your application uses pthreads directly**, you will have to use the "Linux-portable" methods (taskset or numactl) described below.
- **If your application uses OpenMP**, you can use the OpenMP runtime controls which are generally a lot nicer and more powerful.

3.1. The Linux-Portable Way (taskset)

If you want to launch a job (e.g., `simulation.x`) on a certain set of cores (e.g., `core0`, `core2`, `core4`, and `core6`), issue

```
$ taskset -c 0,2,4,6 simulation.x
```

If your process is already running, you can define thread affinity while in flight. It also lets you bind specific TIDs to specific processors at a level of granularity greater than specifying `-c 0,2,4,6` because Linux may still schedule two threads on `core2` and nothing on `core0`. For example,

```
$  for i in $(pgrep application.x);do ps -mo pid,tid,fname,user,psr -p $i;done
  PID   TID COMMAND  USER     PSR
21654     - applicat glock      -
    - 21654 -        glock      0
    - 21655 -        glock      2
    - 21656 -        glock      2
    - 21657 -        glock      6
    - 21658 -        glock      4

$  taskset -p -c 0 21654
$  taskset -p -c 0 21655
$  taskset -p -c 2 21656
$  taskset -p -c 4 21657
$  taskset -p -c 6 21658
```

This sort of scheduling will happen under certain conditions, so specifying a set of cpus to a set of threads without specifically assigning each thread to a physical core may not always behave optimally.

3.2. The Other Linux-Portable Way (numactl)

The emerging standard for easily binding processes to processors on Linux-based supercomputers is `numactl`. It can operate on a coarser-grained basis (i.e., CPU sockets rather than individual CPU cores) than taskset (only CPU cores) because it is aware of the processor topology and how the CPU cores map to CPU sockets. Using `numactl` is typically easier–after all, the common goal is to confine a process to a numa pool (or “cpu node”) rather than specific CPU cores. To that end, `numactl` also lets you bind a processor’s memory locality to prevent processes from having to jump across NUMA pools (called “memory nodes” in `numactl` parlance).

Whereas if you wanted to bind a specific process to one processor socket with `taskset` you would have to

```
$  taskset -c 0,2,4,6 simulation.x
```

the same operation is greatly simplified with `numactl`:

```
$  numactl --cpunodebind=0 simulation.x
```

If you want to also restrict `simulation.x`’s memory use to the numa pool associated with cpu node 0, you can do

```
$  numactl --cpunodebind=0 --membind=0 simulation.x
```

or just

```
$  numactl -C 0 -m 0 simulation.x
```

You can see what *cpu nodes* and their corresponding *memory nodes* are available on your system by using `numactl -H`:

```
$  numactl -H
available: 2 nodes (0-1)
node 0 size: 32728 MB
node 0 free: 12519 MB
node 1 size: 32768 MB
node 1 free: 16180 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

`numactl` also lets you supply specific cores (like `taskset`) with the `–physcpubind` or `-C`. Unlike `taskset`, though, `numactl` does not appear to let you change the CPU affinity of a process that is already running.

An alternative syntax to `numactl -C` is something like

```
$  numactl -C +0,1,2,3 simulation.x
```

By prefixing your list of cores with a `+`, you can have numactl bind to *relative* cores. When combined with cpusets (which are enabled by default for all jobs on Gordon), the above command will use the 0th, 1st, 2nd, and 3rd core of the job's given cpuset instead of literally core 0,1,2,3.

3.3. Using Non-Standard OpenMP Runtime Extensions

Multithreaded programs compiled with Intel Compilers can utilize Intel's Thread Affinity Interface (http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_openmp_thread_affinity.htm) for OpenMP applications. Set and export the *KMP_AFFINITY* env variable to express binding preferences. *KMP_AFFINITY* has three principal binding strategies:

- `compact` fills up one socket before allocating to other sockets
- `scatter` evenly spreads threads across all sockets and cores
- `explicit` allows you define exactly which cores/sockets to use

Using `KMP_AFFINITY=compact` will preferentially bind all your threads, one per core, to a single socket before it tries binding them to other sockets. Unfortunately, it will start at `socket0` regardless of if other processes (such as another SMP job) is already bound to that socket. You can explicitly specify an offset to force the job to bind to a specific socket, but you need to know exactly what is running on what cores and sockets on your node in order to specify this in your submit script.

You can also explicitly define which cores your job should use. Combined with a little knowledge of your system's CPU topology ([Intel's Processor Topology Enumeration tool][intel's processor enumeration tool] is great for this). If you wanted to run on cores 0, 2, 4, and 6, you would do

```
export KMP_AFFINITY='proclist=[0,2,4,6],explicit'
```

GNU's implementation of OpenMP has a environment variable similar to *KMP_AFFINITY* called *GOMP_CPU_AFFINITY*. Incidentally, Intel's OpenMP supports *GOMP_CPU_AFFINITY*, so using this variable may be a relatively portable way to specify thread affinity at runtime. The equivalent *GOMP_CPU_AFFINITY* for the *KMP_AFFINITY* I gave above would be:

```
export GOMP_CPU_AFFINITY='0,2,4,6'
```

3.4. Using OpenMP 4.0 Runtime Controls

Because `KMP_AFFINITY` and `GOMP_CPU_AFFINITY` turn out to be essential for good performance in OpenMP applications, the OpenMP 4.0 standard introduced a few environment variables to accomplish the same effect in a portable, standardized way:

- `OMP_PROC_BIND` allows you to specify that you want to prevent threads from migrating between cores, and optionally, the binding strategy you wish to use
- `OMP_PLACES` allows you specify a more explicit binding strategy

Specifically, `OMP_PROC_BIND` can be set to one of the following values:

- `true` - don't prescribe a specific binding strategy, but when a thread is launched on a core, don't ever let it migrate to another core

- `spread` - Evenly spread threads across all sockets and cores. This is the same as `KMP_AFFINITY=scatter` above.
- `close` - Fill up one socket with threads before allocating threads to other sockets. This is the same as `KMP_AFFINITY=compact`

The `OMP_PLACES` environment variable allows you to specify groups of cores on which you'd like your threads to be bound. This allows you to run an OpenMP application on only a subset of the processor cores available. For example,

- `OMP_PLACES='sockets(1)'` - only allow the application to run on the cores provided by a single CPU socket
- `OMP_PLACES='cores(4)'` - only allow the application to run on the hardware threads provided by four CPU cores. If each CPU core has multiple hardware threads (e.g., Intel HyperThreading), the application can still use all of the hyperthreads on all of the cores.
- `OMP_PLACES='threads(16)'` - only allow the application to run on sixteen hardware threads

You can also use `OMP_PLACES` to achieve the same effect as `KMP_AFFINITY=explicit`. For example,

- `OMP_PLACES='{0},{2},{4},{6}'` will allow OpenMP threads to run on cores 0, 2, 4, and 6.
- `OMP_PLACES='{0,1},{2,3},{4,5},{6,7}'` will allow OpenMP threads to bind to four distinct places, where each place consists of two cores. OpenMP threads will then be able to migrate between the two cores in each place, but not between places.

There are more options and details than what I've listed, and NERSC has a good explanation of how to use OpenMP 4.0 Thread Affinity Controls (https://www.nersc.gov/users/software/programming-models/openmp/process-and-thread-affinity/). However, the easiest way to get a handle on exactly what these options do on your nodes is to experiment with them. Here is a very simple OpenMP script that reports on how threads are allocated to different cores that is very helpful to this end:

```
#include <stdio.h>
#include <sched.h>

int main( int argc, char**argv )
{
#pragma omp parallel
    {
        printf( "Hello world from thread %d of %d running on cpu %2d!\n",
            omp_get_thread_num()+1,
            omp_get_num_threads(),
            sched_getcpu());
    }
    return 0;
```

lennklockwood/79262be4403c341b858deff1f94fa9b8/raw/185367737473086111c4aa6eb8590c2c2d40b757/omp_binding.c)
omp_binding.c (https://gist.github.com/glennklockwood/79262be4403c341b858deff1f94fa9b8#file-omp_binding-c) hosted with ❤ by **GitHub (https://github.com)**

Try building it and running it with different values of `OMP_PROC_BIND` and `OMP_PLACES`.

3.5. getfreesocket

I wrote a small perl script called `getfreesocket` (https://github.com/glennklockwood/sdsc/blob/master/getfreesocket) that uses `KMP_AFFINITY=explicit` (or *GOMP_CPU_AFFINITY*) and some probing of the Linux OS at runtime to bind SMP jobs to free processor sockets. It should be invoked in a job script something like this:

```
#!/bin/bash

NPROCS=1
BINARY=${HOME}/bin/whatever

nprocs=$(grep '^physical id' /proc/cpuinfo  | sort -u | wc -l)
ncores=$(grep '^processor' /proc/cpuinfo | sort -u | wc -l)
coresperproc=$((ncores/nprocs))
OMP_NUM_THREADS=$((NPROCS*coresperproc))

freesock=$(getfreesocket -explicit=${NPROCS})
if [ "z$freesock" == "z" ]
then
  echo "Not enough free processors!  aborting"
  exit 1
else
  KMP_AFFINITY="granularity=fine,proclist=[$freesock],explicit"
  GOMP_CPU_AFFINITY="$(echo $freesock | sed -e 's/,/ /g')"
fi

export KMP_AFFINITY OMP_NUM_THREADS GOMP_CPU_AFFINITY

${BINARY}
```

This was a very simple solution to get single-socket jobs to play nicely on the shared batch system we were using at the Interfacial Molecular Science Laboratory (http://glass.rutgers.edu). While `numactl` is an easier way to accomplish some of this, it still requires that you know what other processes are sharing your node and on what CPU cores they are running. I've experienced problems with Linux's braindead thread scheduling (http://glennklockwood.blogspot.com/2012/07/braindead-thread-scheduling-in-linux.html) so this `getfreesocket` finds completely unused sockets that can be fed into `taskset`, *KMP_AFFINITY*, or `numactl`.

This is not as great an issue if your resource manager supports launching jobs within cpusets. Your resource manager will provide a cpuset, and using relative specifiers for numactl cores (e.g., `numactl -C +0-3`) will bind to the free socket provided by the batch environment. Of course, this will not specifically bind one thread to one core, so using *KMP_AFFINITY* or *GOMP_CPU_AFFINITY* may remain necessary.

Last modified 9/18/2017 at 8:45 PM PDT

glenn@glennklockwood.com (mailto:glenn@glennklockwood.com)