ACRC How-To: Running Large Memory Jobs on BlueCrystal Phase 3

Table of Contents

Introduction

The majority of compute nodes in BlueCrystal phase 3 contain 64GB of RAM and 16 processor cores. That works out at 4GB/core. This is fine for the majority of jobs that run on the cluster. However there will be some jobs that require more memory per core. To accommodate these jobs, BlueCrystal phase 3 also contains a number of large-memory nodes, each with 256GB of RAM. These nodes are accessed via the **himem** queue. (Type **qstat -q** to see all available queues.)

Ordinarily this is where provision for large-memory jobs would stop. However, BlueCrystal phase 3 can support jobs with even larger memory footprints using a technology called **ScaleMP**. This product allows us to combine nodes together such that they appear to be a single node but with a memory capacity equal to the combined RAM of the subsumed nodes. For example, taking 4 of the large-memory nodes and combining them together yields a node with an apparent RAM of around 1TB (the total will be a little less than 4 x 256GB due to some overheads). *Virtual* large memory nodes created using ScaleMP can be accessed via the **vsmp** queue. At the date of writing the sSMP is configured to have ~2.5TB of RAM.

In the following sections I'll describe how to run large-memory jobs (serial, multi-threaded and MPI), complete with example submission scripts, on each of the 3 types of node—standard, high-memory and vSMP.

Running Large Memory Jobs on Standard Nodes

As mentioned in the introduction, the standard compute nodes in BlueCrystal phase 3 contain 64GB of RAM and 16 cores, giving a nominal 4GB of RAM for each core. I say nominal, however, as the RAM in a single node is a shared resource for all the cores in that node and there are no barriers segregating the memory *"belonging"* to one core from that of another. In the absence of any such partitioning, we must all be good citizens and request the resource that we actually need to run our jobs. The upside to this arrangement is flexibility. For example, we can arrange for a core to use more than 4GB by creating an appropriate request.

Serial Jobs

Imagine that we have a serial job which we want to run on a standard compute node. The program does not require more than 4GB of RAM. In this case, the appropriate resource to request in our submission script is a single core:

```
#!/bin/bash
#PBS -l nodes=1:ppn=1
…
./my_serial_prog.exe
```

Now, let's imagine that we have another serial job. This time the program requires 8GB of RAM in order to run successfully. To accommodate it, we must modify our resource request. This time we will request 2 cores on a given node:

```
#!/bin/bash
#PBS -l nodes=1:ppn=2
…
./my_serial_prog.exe
```

Note that we still run a serial job, which will run on a single core. The resource request that we lodge with the queuing system and the actual number of cores that we run on are separate things and are both under our control. By requesting two cores, we have secured ourselves access to two cores worth of RAM—8GB in total. Anyone else running a job on the same node as your job will be able to enjoy the benefits of a maintained ratio of 4GB/core. The same logic applies in 4GB chunks up to the 64GB total RAM for a standard node; if you need 32GB, request 8 cores; if you need 64GB, request the full node.

Multi-threaded Jobs

The logic of resource requests for serial jobs applies in exactly the same way for multi-threaded jobs. Let's imagine that we have an OpenMP program that requires 8GB per thread and that we wish to use 2 threads. In this case we'll need to formulate our resource request such that we reserve 16GB of RAM—four cores worth. We can control the number of threads spawned by setting the OMP_NUM_THREADS environment variable in our submission script:

```
#!/bin/bash
#PBS -l nodes=1:ppn=4
…
export OMP_NUM_THREADS=2
./my_openmp_prog.exe
```

Similarly, if we needed 16GB/thread, we could request the full node and run, 4 threads:

```
#!/bin/bash
#PBS -l nodes=1:ppn=16
…
export OMP_NUM_THREADS=4
./my_openmp_prog.exe
```

MPI Jobs

In principle, with some deft manipulation of the machine file created in your submission script, you can apply the same control over the amount of RAM available to each of the processes in your MPI job. However, in the vast majority of cases this would be an unnecessary complication. This is because MPI programs are typically designed around a principle of data domain decomposition. Each of the processes in an MPI job typically stores and works on a fraction of the overall data. For example, each MPI rank may only be concerned with a portion of the cells contained within a larger, overall grid. Thus, if you find that your MPI job is exceeding the RAM of a standard node, you can usually remedy this by simply requesting more nodes in your submission script and leaving the nodes fully populated with processes.

Checking How Much Memory is Used by Your Job

It is important that we check how much memory our jobs are actually using for two reasons:

- We don't want to encroach on memory that is required by other peoples jobs, and
- Exceeding the limits of RAM on a particular node will result in very poorly performing processes.

Once a job is running on the cluster, we can see how much memory it is using, and how well it is running using the **top** command. First, we must discover which node(s) a job is running on. You can use:

```
qstat -u <your-username>
```

replacing <*your-username*> with, in my case ggdagw, to see what jobs you have running on the cluster.

Then you can use:

```
qstat -n <jobid>
```

replacing <*jobid*> with an appropriate value for one of your jobs, to find which node(s) it is running on. For example:

```
[ggdagw@newblue2 ~]$ qstat -n 832192
...
   node46-009/0+node46-009/1+node46-009/2+node46-009/3+node46-009/4
   +node46-009/5+node46-009/6+node46-009/7+node46-009/8+node46-009/9
   +node46-009/10+node46-009/11+node46-009/12+node46-009/13+node46-009/14
   +node46-009/15
```

indicates a job with allocated 16 cores, all on the same node, namely **node46-009**.

Then you can ssh to the listed node(s) and run **top**. For example:

```
Tasks: 475 total,   2 running, 473 sleeping,   0 stopped,   0 zombie
Cpu(s):  6.5%us,  0.0%sy,  0.0%ni, 93.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  66046116k total,  8519432k used, 57526684k free,   217720k buffers
Swap: 11999224k total,    19828k used, 11979396k free,  5230276k cached

  PID  USER     PR  NI  VIRT   RES  SHR S %CPU %MEM    TIME+  COMMAND
 39043 ggdagw   20   0   1911m 1.9g  320   R 100.0   3.0         0:19.19  foo.exe
...
```

In this first case, I am running a serial executable called **foo.exe**. The program (written in C) allocates a single, large array of integers and randomly accesses various cells in that array. We can see that the program uses approximately 2GB (out of a total of 64GB—I requested the whole node). We can see that the program is performing well as the %CPU column is 100, indicating the that program is able to utilise the CPU to the maximum.

Next, I increased the memory demands of the program, and allocated 80GB of memory. Running top a second time, we see:

```
Tasks: 475 total,   2 running, 473 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  1.7%sy,  0.0%ni, 93.1%id,  5.2%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  66046116k total, 47509104k used, 18537012k free,     2552k buffers
Swap: 11999224k total,  4147976k used,  7851248k free,    82268k cached

  PID  USER     PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 44801 ggdagw   39  19   80.0g  44g  320   R  25.5     70.1         0:18.38   foo.exe
...
```

80GB is well over the 64GB limit of a standard node and we can see that the performance of the program is significantly degraded. In this case, it can only utilise 25% of the CPU. Any program in this situation **will run very slowly** and consequently will take much more time to complete than another which is able to better utilise the CPU.

(The difference between the VIRT and RES columns shows that the operating system has done its best to limit the amount of memory required by the process. Despite the operating system's best efforts, the performance of the process is very poor.)

NUMA and Processor Affinity

At this point it is useful to introduce the concept of **processor affinity**, where a thread or process may be assigned to a particular processor (or range of processors). Ordinarily the operating system is at liberty to schedule threads or processes on the processor core of its pleasing. Indeed, the threads or processes can even be moved from core to core as the program progresses, if the operating system deems it. This behaviour can be fine for some programs, but can severely limit the performance of others.

The reason why some programs can perform badly when subject to arbitrary core selection is down to a memory design principle called **NUMA**, which stands for **N**on-**U**niform **M**emory **A**ccess. Most modern computers attempt to avoid memory access bottlenecks by adopting a NUMA design. A consequence, however, is that portions of the overall memory can be more readily accessed by some cores than by others. Since memory access is often a significant factor in the performance of a program the matching of cores to banks of memory can be crucial for best performance. In cases where some cores share a level of cache, and where the program has a particularly high memory bandwidth requirement, it can also be beneficial to spread threads or processes out over a sparsely utilised pool of cores.

Happily processor affinity can be set using one of several tools at our disposal. I will introduce one of several options in turn.

The first tool is called **taskset** and is available for most Linux distributions. You can use it to pin threads (or processes) to cores. From a portability perspective, it has the advantage of being widely available but has the disadvantage that cores IDs must be explicitly specified—what if you switched machines and the core topology or core count was different?

You can discover the how core IDs map onto the physical makeup of you machine by consulting **/proc/cpuinfo**. In that file, **physical id** refers to the socket. The standard compute nodes on BlueCrystal phase 3 have two sockets, each supporting 8 cores—16 cores in total. Cores 0-7 are on one socket and 8-15 are on the other. (Note that the login nodes of the cluster also have two sockets and 16 cores, but that the assignment is different—even core IDs on one socket, odd IDs on the other.)

As an example of its use, in the submission script below we specify that the 8 threads must be run on cores 0 through to 7 (i.e. resident on a single socket):

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
…
export OMP_NUM_THREADS=8
taskset -c 0-7 ./my_openmp_prog.exe
```

Alternatively, we could specify that two threads must be run on cores located on different sockets:

```
#!/bin/bash
#PBS -l nodes=1:ppn=2
…
export OMP_NUM_THREADS=2
taskset -c 0,8 ./my_openmp_prog.exe
```

A word of caution, however. When using taskset we must explicitly pin threads (or processes) to individual cores. If another user on the same machine also explicitly sets processor affinity, then there is a risk that the operating system will have no choice but to schedule two threads on the same core, with a resulting catastrophic loss of performance, which would nullify again gains that you had made in aligning processors and memory banks.

Fortunately, there is a Linux portable alternative to explicitly pinning threads to cores. **numactl** allows you to bind threads to sockets. This looser arrangement gives the operating system scope to load balance. (You can still pin explicitly to cores using numactl, should you wish.)

A portion of a submission script using numactl instead of to place threads on a single socket (ID 0) and to ensure that only the memory directly associated with that socket is used would look like:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
…
export OMP_NUM_THREADS=8
numactl --cpunodebind=0 --membind=0 ./my_openmp_prog.exe
```

numactl --hardware is a useful command for discovering the core and socket topology of a machine. Both taskset and numactl have useful man pages.

A third way to specify *thread* affinity is the use of the **KMP_AFFINITY** environment variable. **This can only be used with OpenMP code compiled with an Intel compiler, however**. By setting the value of the environment variable appropriately, you have the option to explicitly pin to cores, or to specify looser patterns of affinity, such as a **compact** grouping or to **scatter** widely.

An example of requesting that threads be placed as close together as possible using KMP_AFFINITY in a submission script is:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
…
export OMP_NUM_THREADS=8
export KMP_AFFINITY=compact,verbose
./my_openmp_prog.exe
```

To scatter the threads evenly over the whole machine you would use:

- **export KMP_AFFINITY=scatter,verbose**

Or to explicitly pin to cores:

- **export KMP_AFFINITY='proclist=[0,2,4,6],explicit,verbose'**

Adding **verbose** to the option list is useful if you want to see where the threads were run.

OpenMP code compiled using the GNU compiler can use a similar environment variable to pin explicitly to cores, e.g.:

- **export GOMP_CPU_AFFINITY='0,2,4,6'**

Running Large Memory Jobs on High Memory Nodes

BlueCrystal phase 3 contains 16 nodes high-memory nodes, each of which has 256GB of RAM. You can use these nodes if your program cannot be run on the standard compute nodes. However, be aware that since there relatively few of them (compared to more than 300 standard compute nodes) your jobs on the high-memory nodes may be queued for longer as the competition for them may be stronger.

All of the details from the previous section on standard compute nodes—resource requests, controlling the number of threads, processor affinity etc.—are transferable to jobs run on the high-memory nodes. The only difference is that the jobs should be submitted to the **himem** queue. A submission script for a serial job which required a full 256GB of would look like:

```
#!/bin/bash
#PBS -q himem
#PBS -l nodes=1:ppn=16
…
./my_serial_prog.exe
```

If you are compelled to use the high-memory nodes due to the RAM requirements of your program but are concerned about queuing time, it could be worthwhile searching for alternative software that makes more intelligent use of standard compute nodes. The mpiBLAST program is an example of some software that can efficiently perform a typically high-memory task on generic computational hardware.

Running Large Memory Jobs on a vSMP Node

Using ScaleMP technology, we can combine high-memory nodes in BlueCrystal phase 3 to create virtual nodes with even larger memories. For example, four high-memory nodes can be combined to create a node with around 1TB of RAM, twelve would yield a vSMP with ~3TB of memory. The vSMP is a very flexible device. It can be used to run serial jobs, multi-threaded jobs and even MPI jobs, should the need arise. The VSMP can support multiple jobs running at the same time. The queuing system will apportion only as much of the vSMP as is requested in the submission script. Example submission scripts are given in the sections below. However, since we have only one vSMP, competition for this resource may be stronger than for other nodes in the cluster.

Serial Jobs

The simplest scenario that I can think of is the need to run a serial program with a very large memory requirement. Below is an example submission script:

```
#!/bin/bash

#PBS -q vsmp
#PBS -l nodes=1:ppn=32

cd $PBS_O_WORKDIR

export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so

CORES=/dev/cpuset/torque/${PBS_JOBID}/cpus
# Torque lists the cores as a range, e.g. 0-31
# The perl oneliner turns that range into a list.  We take the first core to set cpu affinity
FIRST=`cat $CORES | perl -nle '@a=split(/-/); foreach ($a[0]..$a[1]) {print "$_"}' | head -1`

time taskset -c ${FIRST} ./my_very_large_serial_job.exe
```

Looking at each of the lines relevant to our very large memory job:

- The vSMP nodes have their own queue and **#PBS -q vsmp** ensures that the job is submitted to the correct queue.
- Although we are only using one processor in this case, we must **remember to request sufficient memory to run the job**. The smallest unit of resource we can reserve on a vSMP node is a socket, which supports 8 cores for BlueCrystal phase 3. This equates to 128GB of RAM. You should request 16 cores for 256GB of RAM, 32 cores for 512GB of RAM etc. You must also allow some margin—around 12.5%—since the vSMP infrastructure itself requires some memory in which to operate. So, for example, if we request 32 cores worth of RAM we would expect to be able to run a job which requires (512 - 64) 448GB of RAM. **If your job consumes more memory than that requested for it, it will be killed by the queuing system**.
- The **export LD_PRELOAD=...** line loads a special library which cuts out some system calls that are unnecessary on a vSMP node.
- As with other machines, **NUMA effects are important on the vSMP**. Indeed, given that some portions of memory are accessed via a network cable, cpu affinity becomes even more important. The queuing system will keep a record of the cores assigned to a job run on the vSMP. If all the cores are contiguous, this information will be stored as a range, e.g. 0-31. We would like to set the cpu affinity for our serial job and I have arbitrarily chosen to pin the computation to the first core in the given range. The **FIRST** environment variable and it's associated perl 'one-liner' captures that core ID (after first turning the range into an explicit list of cores), so that it can be used with tasket, on the last line.

Multi-threaded Jobs

Of course, we are not limited to running serial jobs on the vSMP. Perhaps its greatest use will be to run very memory hungry multi-threaded jobs. Here is another example submission script which we can use to run an OpenMP program:

```
#!/bin/bash

#PBS -q vsmp
#PBS -l nodes=1:ppn=32

cd $PBS_O_WORKDIR

export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so
export OMP_NUM_THREADS=32

./my_very_large_openmp_job.exe
```

You will notice that after all the talk of cpu-affinity, no attempt has been made to pin threads to cores in the above submission script. The reason for this is that the queuing system will assign the most compact set of cores to the job. This being the case, adding a line such as, *KMP_AFFINITY=compact*, will have no additional effect. However, if you will to use fewer cores that the amount requested, and wish to spread them around evenly (and you have compiled using Intel), you will want to use *KMP_AFFINITY=scatter*.

If your job makes use of Intel's Math Kernel Library (MKL), there is an additional environment variable, MKL_VSMP, that will give you some additional performance:

```
#!/bin/bash

#PBS -q vsmp
#PBS -l nodes=1:ppn=32

module add intel-cluster-studio/compiler/64/13.1/117
module add intel-cluster-studio/mkl/64/13.1/117

cd $PBS_O_WORKDIR

export LD_PRELOAD=/opt/ScaleMP/libvsmpclib/0.1/lib64/libvsmpclib.so
export OMP_NUM_THREADS=32
export MKL_VSMP=1

./my_very_large_threaded_MKL_job.exe
```

The reason for this is that setting MKL_VSMP=1 causes the library to use memory in a way which is better suited to the vSMP makeup.

Summary

In this *how-to* document, I have attempted to provide a practical guide to running large-memory jobs on BlueCrystal phase 3, where the term *large memory* will vary in absolute terms, depending upon context. I have shown you how to request (and check that you have) sufficient memory resources for your job, so that it can peacefully co-exist with other jobs running on the cluster. When providing example submission scripts, I have focussed on scenarios where you wish to run serial or multi-threaded programs. The reason for this is that for many MPI programs, the solution to memory constraints is often to simply run your job using more cluster nodes. The topic of non-uniform memory access (NUMA) was also introduced. Most modern computational environments (including the standard two-socket compute nodes on BlueCrystal pahase 3) fall into the NUMA category and setting cpu-affinity appropriately is often important to achieve best performance. Several methods for setting cpu-affinity were introduced and examples given. Lastly, the guide provided examples for submitting jobs to three categories of compute node found in the cluster—standard nodes, large memory nodes and the vSMP node. I hope you found it useful!