Checking memory use

All compute nodes have a limited amount of physical memory – *RAM* – available to your application. If your program exceeds the available memory on a Derecho node, it will either be killed by system monitors or crash and dump core, often with a "bus error" message in your logs. On the Casper cluster, nodes can *swap* data to NVMe storage, which typically prevents failures because of running out of memory in all but the most intensive workflows. Computing on swapped data, however, is much slower than processing data that is resident in memory, so jobs may can still fail from exhausting the requested wall-clock.

Applications that offload computation to GPUs must also consider the available memory on each GPU, which is known as the *VRAM*. Asking for a large amount of memory in every job might seem like a good idea, but it will typically result in slow dispatch times and long queue waits.

Always try to match your memory request to your needs. This allows the scheduler to best fit all jobs onto the available resources. A good match will also prevent you from needlessly waiting for more memory than you actually require.

As detailed below, you can query the PBS scheduler for bulk memory statistics from any completed job. You can also observe memory usage via either instrumenting your application or live monitoring.

Page contents

Available memory by system

System	Usable memory per compute node
Derecho	240 GB (2,488 CPU nodes) 490 GB (82 GPU nodes)
Cheyenne	45 GB (3,168 nodes) 109 GB (864 nodes)
Casper	365 GB (22 nodes) 738 GB (4 nodes) 1115 GB (6 nodes)

If your job approaches the usable memory per node threshold shown in the table, you may experience unexpected issues or job failures. Leave a margin of 2 or 3 percent.

Querying memory usage from PBS

The **qhist** tool can be used to display information about completed PBS jobs. By default, it will print the node memory the job consumed (the high-water mark usage), as in this example:

```
$ qhist -u $USER -d 3
Job ID    User       Queue    Nodes NCPUs NGPUs Finish    Mem(GB)   CPU(%)  Elap(h)
3632559   benkirk    htc          1     1     0 27-1802      21.8     99.0     8.01
3632475   benkirk    htc          1     1     0 27-0949      30.1     86.0     0.26
...
```

The output shows all jobs run in the previous three days, and it lists the amount of memory consumed.

Running qhist with the **-w** argument specifies "wide" output and provides additional details, including the memory requested, as in this example:

```
$ qhist -u $USER -d 3 -w
Job ID    User            Queue      Nodes NCPUs NGPUs Finish Time  Req Mem Used Mem(GB)  Avg CPU (%)  Elapsed
(h) Job Name
3632559   benkirk         htc            1     1     0 07-27T18:
02     40.0         21.8         99.0         8.01 STDIN
3632475   benkirk         htc            1     1     0 07-27T09:
49    256.0         30.1         86.0         0.26 regrid_bmr_casper
...
```

The output in the first job shows that it requested 40 GB of memory but used only 21.8 GB. The second job requested 256 GB but used only 30.1GB.

If you see a job where the used memory matches the requested memory, it likely either failed with an out-of-memory error or began swapping data to disk, negatively affecting performance. Using **qhist**, you can easily check for large under- and overestimates and adjust future jobs accordingly.

PBS logs do not include information about GPU memory usage.

Instrumenting your application

While the scheduler can provide the maximum memory amount used by a job, it cannot provide details on specific commands in your script, time-varying memory logging, or insight into GPU VRAM usage. For these details, you will want to turn to application instrumentation. Using instrumentation will incur some memory usage overhead, so it can be a good idea to request more memory than your job would typically need when profiling.

Using peak_memusage to report and log memory usage

The **peak_memusage** tool is a command-line utility that outputs the maximum node memory – and GPU memory, if relevant) – used by each process and /or thread of an application. It has several modes of operation which vary in detail reported and complexity of use.

peak_memusage for determining maximum memory usage

The peak_memusage command can be used to run your application and report its "high water," or peak memory footprint. To use it, you must load the appropriate module and simply prefix your application launch command as follows:

```
$ module load peak-memusage
$ peak_memusage <my_application> <--my-apps-arguments>
...
casper-login1 used memory in task 0: 51MiB (+2MiB overhead). ExitStatus: 0. Signal: 0
```

The regular application output has been abbreviated. The final line of output reveals this application required a total of 51MiB of system memory, including 2MiB "overhead." The overhead may vary considerably by application and depends largely on what libraries the application requires. This approach also works inside MPI applications, with specific examples provided here. Run **man peak_memusage** for additional details and examples.

Determining memory use history

CISL provides a **log_memusage** tool you can use to produce a time history of system memory usage as well as GPU memory if allocated to the job. This tools provides a shared library that can be either linked to directly or, more likely, used with an existing application through the **LD_PRELOAD** mechanism. For example, if you have a previously compiled application and would like to determine its memory usage over time, set the environment variable LD_PREL OAD when executing the application as follows:

```
$ LD_PRELOAD=${NCAR_ROOT_PEAK_MEMUSAGE}/install/lib/liblog_memusage.so <my_application>
...
# (memusage) --> gust02 / PID 44899, peak used memory: 18 MiB
```

The **log_memusage** tool works by creating a *monitoring thread* that polls the operating system for the application's memory usage at specified intervals. Several environment variables control the tool's behavior. Run **man log_memusage** for additional details and examples. This approach also works under MPI if the **LD_PRELOAD** environment variable is propagated properly into the MPI execution, as in this example:

```
$ mpiexec -n 6 -env LD_PRELOAD=${NCAR_ROOT_PEAK_MEMUSAGE}/lib/liblog_memusage.so <my_application>
...
# (memusage) --> gu0017 / PID 121027 / MPI Rank 0, peak used memory: 166 MiB (CPU), 563 MiB (GPU)
# (memusage) --> gu0017 / PID 121029 / MPI Rank 2, peak used memory: 93 MiB (CPU), 563 MiB (GPU)
# (memusage) --> gu0017 / PID 121031 / MPI Rank 4, peak used memory: 93 MiB (CPU), 563 MiB (GPU)
# (memusage) --> gu0017 / PID 121032 / MPI Rank 5, peak used memory: 92 MiB (CPU), 563 MiB (GPU)
# (memusage) --> gu0017 / PID 121030 / MPI Rank 3, peak used memory: 93 MiB (CPU), 563 MiB (GPU)
# (memusage) --> gu0017 / PID 121028 / MPI Rank 1, peak used memory: 165 MiB (CPU), 563 MiB (GPU)
```

The details for how a given MPI implementation passes environment variables can vary. Some use **-env** while others use **-x**. Run **man mpiexec** for specifics of your implementation.

Using cray-mpich to report and log Derecho memory

The preferred Cray MPICH MPI implementation on Derecho provides capabilities, similar to those above, for recording an application's high-water memory usage. They are controlled by these environment variables:

- **MPICH_MEMORY_REPORT**
- **MPICH_MEMORY_REPORT_FILE**

Here's an example:

```
export MPICH_MEMORY_REPORT=1
mpiexec <typical arguments> ./myapp
...
# MPICH_MEMORY: Max memory allocated by malloc:  5540920 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by malloc:  5540224 bytes by rank 1
# MPICH_MEMORY: Max memory allocated by mmap:    110720 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by mmap:    110720 bytes by rank 0
# MPICH_MEMORY: Max memory allocated by shmget:  17834896 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by shmget:  0 bytes by rank 1
```

The **mpiexec** command also provides a high-level summary of resources consumed, including memory use, when the **--verbose** flag is employed:

```
mpiexec --verbose <typical arguments> ./myapp
...
Application dc6a4084 resources: utime=7s stime=7s maxrss=222296KB inblock=0 oublock=0 minflt=289502 majflt=0
nvcsw=16550 nivcsw=276
```

The **maxrss** (maximum resident set size) is the maximum CPU memory required by a single MPI rank in the application. Run **man intro_mpi** and **man mpiexec** on Derecho for more information on these Cray-specific features.

Other parallel memory profiling tools

We recommend the tools described above for understanding your job memory use and crafting your PBS resource specifications. However, sometimes you need even more detailed information on application memory use patterns (e.g., during code development cycles). Many tools exist for detailed memory profiling; on Derecho and Casper we recommend Arm Forge.

Interactive monitoring

The tools described above record the memory consumed during a program's execution, but monitoring in real time can be useful, too. It allows you to take action if you need to while the job is running and you have access to the compute nodes. Take care when executing additional commands on your job nodes, as any additional load you put on the execution host will slow your program down, and any additional memory you consume subtracts from what is available. This consideration is especially important on the Casper cluster, where your host is likely shared with other users.

The first step is to identify which nodes are in use by running the **qstat -f <JOBID>** command as shown here:

```
Job Id: 5677216.casper-pbs
...
    resources_used.mem = 78004288kb
    resources_used.ncpus = 1
    resources_used.vmem = 1262344kb
    resources_used.walltime = 00:19:27
    job_state = R
    queue = htc
    server = casper-pbs
...
    exec_host = crhtc53/17*4+crhtc62/11*4
...
    project = _pbs_project_default
    Submit_Host = chadmin4.hpc.ucar.edu
```

The command will create a lot of output, which is abbreviated above. Note specifically that PBS is reporting the amount of memory *used so far*.

The execution hosts (compute nodes) are listed in the *exec_host* field. The job in the example above ran on two hosts, crhtc53 and crhtc62. While the job is in a running state, it is possible to **ssh** directly to any of the named hosts and query basic diagnostic programs such as **top** (for node memory) and **nvidia-smi** (for GPU memory), or use more advanced tools such as **strace** and **gdb**. Typically, you will want to begin with the first execution host listed, as many applications perform file I/O on the first host so memory usage tends to be highest there. Be aware that your ssh session will be terminated abruptly when the PBS job ends and you no longer have batch processes assigned to the host(s).

Understanding and resolving various memory issues

Issue 1. Your Derecho or Casper job shows signs of a memory issue (e.g., the job log ends with "Killed" or "Bus Error") despite **qhist** reporting a higher requested than used memory amount.

The output from **qhist** indicates the total memory used and requested across all nodes in the job. This heuristic, while useful, can obfuscate memory usage patterns. Many scientific applications perform read/write operations on the "head" (primary) node of the job, and because data must be collected to this node to do I/O, memory usage can exceed what is available on that node while others use far less. Additionally, the PBS scheduler only records data at set intervals, so aliasing issues come into play for rapid increases in memory footprint. More sophisticated instrumentation like the temporal logging mode of **peak_memusage** can provide insight into these memory usage patterns.

Issue 2. Your data analysis job on Casper begins to run slowly after a large array has been allocated, but no memory error is reported.

It is likely that you are swapping data from RAM to disk. If possible, perform garbage collection in your language of choice to free memory. Otherwise, you will likely need to start a new job with a higher memory limit.

Issue 3. Your batch job crashes shortly after execution and the output provides no explanation.

If you suspect memory problems, it makes sense to examine recent changes to your code or runtime environment. Ask yourself questions like these:

- When did you last run the job successfully? Was the model's resolution increased since then?
- Did you port your code from a computer having more memory?
- Did you add new arrays or change array dimensions?
- Have you modified the batch job script?

If the answer to those questions is “no,” your job might be failing because you have exceeded your disk quota rather than because of a memory issue. To check, follow these steps:

- Run the **gladequota** command.
- Check the "% Full" column in the command's output.
- Clean up any GLADE spaces that are at or near 100% full.
- Look for core files or other large files that recent jobs may have created.
- Exceeding disk quota at runtime can result in symptoms that are similar to those resulting from memory problems. If running gladequota doesn't indicate a problem, consider using the ARM Forge DDT advanced memory debugger to isolate the problem.

If you have tried all of the above and are still troubled that your job is exceeding usable memory, contact the NCAR Research Computing help desk with the relevant job number(s).