Slurm memory limits

Slurm imposes a memory limit on each job. By default, it is deliberately relatively small — 2 GB per node. If your job uses more than that, you'll get an error that your job *Exceeded job memory limit*.

To set a larger limit, add to your job submission:

```
#SBATCH --mem X
```

where X is the maximum amount of memory your job will use per node, in MB.

The larger your working data set, the larger this needs to be, but the smaller the number the easier it is for the scheduler to find a place to run your job.

To determine an appropriate value look at how to study job efficiency section.

How to study job efficiency

Many times we ask ourselves how I figure out how efficient my job is and for this study you can use the ***seff*** command.

```
seff JOBID
```

where JOBID is the one you're interested in. For example:

```
$ seff 1553

Job ID: 1553
Cluster: teide
User/Group: viddata/viddata
State: COMPLETED (exit code 0)
Nodes: 4
Cores per node: 16
CPU Utilized: 2-11:55:56
CPU Efficiency: 89.73% of 2-18:47:28 core-walltime
Job Wall-clock time: 01:02:37
Memory Utilized: 26.08 GB (estimated maximum)
Memory Efficiency: 23.85% of 109.38 GB (27.34 GB/node)
```

```
seff JOBID
```

Request memory little large than seff reports.

You should set the memory you request to something a little larger than what seff reports.

***seff* show the maximum memory used in parallel jobs.**

Note that **for parallel jobs spanning multiple nodes, this is the maximum memory used on any one node**; if you're not setting an even distribution of tasks per node (e.g. with --ntasks-per-node), the same job could have very different values when run at different times.

Wait for job to finish succesfully

Also note that the number recorded by slurm for memory usage will be inaccurate if the job terminated due to being out of memory. **To get an accurate measurement you must have a job that completes successfully as then slurm will record the true memory peak**.

Max memory per type of node

Node type	Slurm memory max request	
Sandy bridge 16 Cores - 32 GB	30000 MB	
Sandy bridge 16 Cores - 64 GB	62000 MB	
Fat Nodes 32 Cores - 256 GB	254000 MB	upon request
Icelake Nodos GPU 64 Cores - 256 GB	254000 MB	upon request

How do I figure out how efficient my job is?

You can see your job efficiency by comparing MaxRSS, MaxVMSize, Elapsed, CPUTime, NCPUS with *sacct* command.

```
sacct -j 999997 --
format=User,JobID,Jobname%25,partition,elapsed,MaxRss,MaxVMSize,nnodes,ncpus,nodelist%
```

```
    User       JobID         JobName  Partition    Elapsed     MaxRSS  MaxVMSize
NNodes     NCPUS          NodeList
--------- ------------ ---------- ---------- ---------- ---------- -------- ----------
 eolicase 999996         WRF_2023080618      batch   13:32:47                    4       64
node1511-[1-4]
        999996.batch          batch           13:32:47    216700K    815628K      1       16
node1511-1
        999996.0        geogrid.exe           00:00:57    236240K   2762564K      4        8
node1511-[1-4]
        999996.1        metgrid.exe           00:01:54     15395M    591912K      4        4
node1511-[1-4]
        999996.2          real.exe          00:00:26    338148K   2928876K      4       16
node1511-[1-4]
        999996.3           wrf.exe          13:26:36    598864K   3768748K      4       64
node1511-[1-4]
```

In this job, you see that the user used 64 cores and his job ran for 13.5 hours. However, your CPU time is 13.5 hours, which is close to 64*13 hours. If your code scales effectively according to this formula *CPUTime = NCPUS * Elapsed* your application scales perfectly. If it isn't, the result will diverge from the formula and the best way to test this and determine how to scale your app is to do some scaling testing.

There are two styles you can do: ***Strong scaling*** and ***Weak scaling***

Strong scaling

Strong scaling is where you leave the problem size the same but increase the number of cores. If your code scales well it should take less time proportional to the number of cores you use.

Weak scaling

The amount of work per core remains the same but you increase the number of cores, so the size of the job scales proportionally to the number of cores. Thus if your code scales in this case the run time should remain the same.

Typically most codes have a point where the scaling breaks down due to inefficiencies in the code.

Thus beyond that point there is not any benefit to increasing the number of cores you throw at the problem. That’s the point you want to look for. This is most easily seen by plotting log of the number of cores vs. log of the runtime.