IBG Research Computing Cheatsheet

Best practices

We all share resources–learning to effectively manage these resources not only increases the efficiency of your work but that of your colleagues. Here are a few principles to consider when allocating resources.

1. Memory and CPU usage should be proportional.

This principle is best demonstrated via example: the himem nodes `bnode0412`- `bnode0414` have 976gb of RAM and 64 logical cores each. For a job that requires 300gb of RAM, you'll want to request 300/976 * 64 ≈ 20 tasks (`--N 1 --mem 300gb --ntasks 20`; always request cpus in multiples of two, because of hyperthreading slurm will round up anyways). This helps maximize the amount of work we can accomplish simulatenously. E.g., if you were to submit 8 jobs requiring 10gb of RAM and 8 tasks each, you would be using all 64 cores (100% of cpu power) but only 80/976 gb of ram (8% of total memory), rendering the remaining RAM unusuable by anyone. Total memory and cpus vary by node, so you should consider which nodes are best for a given task an use the appropriate constraints (see 'View available nodes and their properties').

2. Use the preemptable QOS when possible

Most of Blanca is idle at any given time–the preemptable QOS provides access to all nodes and has the dual benefits of increasing the amount of computing you can accomplish in a given time frame while freeing speciality hardware (himem nodes) for applications that can only node on those nodes. You should consider the preemptable QOS for any job that requires fewer than 170gb of ram and will complete in less than a day. Situations where you might NOT want/be able to use the preemptable QOS include: jobs that require more than day to finish, jobs that require more than 170gb of memory, debugging small instances of single jobs before submitting many preemptable jobs, small interactive sessions.

3. Prototype/debug before submitting large numbers of jobs

If you're unsure how much memory/how many cpus a particular task will require, make sure you figure it out before submitting many such jobs. Methods for checking memory use are provided in the next session. You can also directly `ssh` into any node where you have currently running jobs and use `top` or `htop` to examine cpu / memory use in real time. If you find that a job is taking longer than you expected, investigate and cancel if necessary rather than waiting for it to time out.

4. Big jobs are fragile

Big here means consisting of many small tasks or consisting of one giant task that could be split up into smaller tasks. To the extent than you are able, you should discretize large tasks into many small tasks. This makes your workflow more fault tolerant (if some step goes wrong, you don't have to rerun everything) and easier to efficiently allocate resources (e.g., if a job that would take four days can be split into multiple jobs that take less than one day, you can use the premptable QOS; this will most likely finish faster while simulatenously freeing speciality resources–i.e. himem nodes–so they'll be available to your colleages). Simple strategies for reducing job fragility include: - avoid using loops within jobs (consider job-arrays instead) - avoid grouping multiple tasks (e.g., GWAS and summary statistic clumping) in a single job - read documentation to determine if jobs can be split into discrete chunks and then do so (e.g. using `--make-grm-part` in GCTA can allow you to run several fast, low memory, preemptable jobs, often simultaneously, instead of one multi-day job that requires a himem node)

5. Ask for help

If you're having difficulty figuring out how to get your jobs to run efficiently (or run at all) don't hesitate to ask for help from your colleagues.

Monitoring Slurm and Job Activity

Monitoring running jobs

View your running jobs

```
squeue -u $USER
```

- Additional fields displayed using the `-l` flag
- Custom output described in man page. E.g., I find `squeue -o "%.12i %.18j %.16q %.8T %.10M %.12l %.24R"` easier to read

View running jobs on a particular qos

```
squeue -q <QOS>
```

Measure performance of completed jobs

The following flags can be combined to suit your needs

Display jobs completed since a particular date

```
sacct -S <MMDD>
```

Display jobs' timing and memory usage

```
sacct -o 'jobid%20,jobname%16,state,elapsed,maxrss'
```

View available nodes and their properties

```
sinfo  --Node -o  "%.12N %.16P %.11T %.4c %.13C %.8e /%.8m %.30f"
```

- `NODELIST` is the name of the node, can be specified when submitting via –nodelist
- `PARTITION` is the name of the partition. You likely will never need to use this unless you want a particular partition of the preemptable queue
- `STATE` - mixed means that some of the CPUS are in use (see below)
- `CPUS(A/I/O/T)` - on a given node: allocated/idle/other/total nodes
- `FREE_MEM / MEMORY` free memory / total memory in MB (divide by 1024 to convert to GB
- `AVAIL_FEATURES` - architectures/instruction sets requestable via the `--constraint` flag

Example jobs

Here are some example jobs as templates–the actual resources requested should be adjusted to fit your needs.

Basics

Interactive jobs

```
sinteractive --qos preemptable --mem 64gb --ntasks 12 --time 2:00:00
```

or

```
sinteractive --qos blanca-ibg --mem 64gb --ntasks 16 --time 2:00:00 --constraint haswell
```

Preemptable job

```
#!/bin/bash
#SBATCH --qos=preemptable
#SBATCH --mem=100gb
#SBATCH --time=<D-HH:MM> or <HH:MM:SS> # must be at most one day
#SBATCH --ntasks=24
#SBATCH --nodes=1
#SBATCH --array=1-22
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a
```

Job arrays

Job arrays provide a clean method for subbmitting collections of jobs and are often preferable to either i. using a loop to submit multiple jobs or ii. using a loop within a single job to accomplish multiple tasks. Note that the resources you request apply to each job individually, not to the collection of jobs. E.g., if each of 10 tasks requires 50gb of memory, you only need to request 50gb, not 500gb.

Job array with a single numeric index

```
#!/bin/bash
#SBATCH --qos=<QOS>
#SBATCH --mem=<MEM>gb
#SBATCH --time=<D-HH:MM> or <HH:MM:SS>
#SBATCH --ntasks=<Number of cpus>
#SBATCH --nodes=1
#SBATCH --array=1-22
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a

CHR=${SLURM_ARRAY_TASK_ID}


ml load <modules>

program <args> \
    --input <inputprefix>_"$CHR" \
    --output <outputprefix>_"$CHR"
```

Job array with a single non-numeric index

This script submit jobs in parallel for jobs with different inputs

```
#!/bin/bash
...
#SBATCH --array=0-2
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a

ii=${SLURM_ARRAY_TASK_ID}

inputArray=(phenoA phenoB phenoC)
```

```
input=${inputArray[$ii]}

ml load <modules>

program <args> \
    --input $input
```

Job array with a multiple simultaneous indices

This script submit jobs in parallel for jobs with different inputs/outputs

```
#!/bin/bash
...
#SBATCH --array=0-2
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a

ii=${SLURM_ARRAY_TASK_ID}

inputArray=(phenoA phenoB phenoC)
outputArray=(outputA outputB outputC)

input=${inputArray[$ii]}
output=${outputArray[$ii]}

ml load <modules>

program <args> \
    --input $input \
    --output $output
```

Job array with a multiple nested non-numeric indices

This script submit jobs in parallel for jobs with arbitrary nested lists of arguments (e.g., each model for each phenotype) using integer arithmetic. If this is unfamiliar, you can google “floor division bash” and “modulo bash”.

```
#!/bin/bash
...
#SBATCH --array=0-5
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a

ii=${SLURM_ARRAY_TASK_ID}

inputArray=(phenoA phenoB phenoC)
modelArray=(modelA modelB)

inputIndex=$(expr $ii % 3)
modelIndex=$(expr $ii / 3)

input=${inputArray[$inputIndex]}
model=${modelArray[$modelIndex]}

ml load <modules>
```

```
program <args> \
    --phenotype "$input" \
    --model $model"
```

You can always double check that you didn't screw something up by running a simple loop (in the shell):

```
inputArray=(modelA modelB)
modelArray=(phenoA phenoB phenoC)

for ii in {0..5}
do
    inputArray=(phenoA phenoB phenoC)
    modelArray=(modelA modelB)
    inputIndex=$(expr $ii % 3)
    modelIndex=$(expr $ii / 3)
    input=${inputArray[$inputIndex]}
    model=${modelArray[$modelIndex]}
    echo input:"$input" model:"$model"
done
```

Job array with a multiple nested indices, one numeric

You can frequently simplify things when one the lists you iterate over is numeric (e.g., each chromsome for each phenotype):

```
#!/bin/bash
...
#SBATCH --array=0-65
#SBATCH -J <jobname>
#SBATCH -o <output dir>/<jobname>_%a

ii=${SLURM_ARRAY_TASK_ID}

inputArray=(phenoA phenoB phenoC)

inputIndex=$(expr $ii / 22)
chrom=$(expr $(expr $ii % 22) + 1)

input=${inputArray[$inputIndex]}

ml load <modules>

program <args> \
    --phenotype "$input" \
    --genotypes <someprefix>_chr"$chrom"
```

Again, you can check that this works via a loop:

```
inputArray=(phenoA phenoB phenoC)

for ii in {0..65}
do
    inputIndex=$(expr $ii / 22)
    chrom=$(expr $(expr $ii % 22) + 1)
    input=${inputArray[$inputIndex]}
    echo chrom:"$chrom" pheno:"$input"
```

```
done
```

Defining custom functions

Commands that are lengthy to type in or that you frequently used can be turned into functions to save time. To do so, add something along the following lines to `~/.my.bashrc`

```
function Sinfo() {
sinfo  --Node -o  "%.12N %.16P %.11T %.4c %.13C %.8e /%.8m %.30f"
}
export -f Sinfo
```

Then call `source ~/.my.bashrc` and you should be able to use `Sinfo` in place of the longer command above.