

# Capacity computing

## Introduction

In many cases, it is useful to submit huge (>100+) number of computational jobs into the PBS queue system. Huge number of (small) jobs is one of the most effective ways to execute embarrassingly parallel calculations, achieving best runtime, throughput and computer utilization.

However, executing huge number of jobs via the PBS queue may strain the system. This strain may result in slow response to commands, inefficient scheduling and overall degradation of performance and user experience, for all users. For this reason, the number of jobs is **limited to 100 per user, 1500 per job array**

Please follow one of the procedures below, in case you wish to schedule more than >100 jobs at a time.

- Use Job arrays when running huge number of multithread (bound to one node only) or multinode (multithread across several nodes) jobs
- Use GNU parallel when running single core jobs
- Combine GNU parallel with Job arrays when running huge number of single core jobs

## Policy

1. A user is allowed to submit at most 100 jobs. Each job may be a job array.
2. The array size is at most 1000 subjobs.

## Job arrays

Huge number of jobs may be easily submitted and managed as a job array.

A job array is a compact representation of many jobs, called subjobs. The subjobs share the same job script, and have the same values for all attributes and resources, with the following exceptions:

- each subjob has a unique index, \$PBS\_ARRAY\_INDEX
- job Identifiers of subjobs only differ by their indices
- the state of subjobs can differ (R,Q,...etc.)

All subjobs within a job array have the same scheduling priority and schedule as independent jobs. Entire job array is submitted through a single qsub command and may be managed by qdel, qalter, qhold, qrls and qsig commands as a single job.

## Shared jobscript

All subjobs in job array use the very same, single jobscript. Each subjob runs its own instance of the jobscript. The instances execute different work controlled by `$PBS_ARRAY_INDEX` variable.

Example:

Assume we have 900 input files with name beginning with “file” (e. g. file001, ..., file900). Assume we would like to use each of these input files with program executable `myprog.x`, each as a separate job.

First, we create a tasklist file (or subjobs list), listing all tasks (subjobs) - all input files in our example:

```
$ find . -name 'file*' > tasklist
```

Then we create jobscript:

```
#!/bin/bash
#PBS -A PROJECT_ID
#PBS -q qprod
#PBS -l select=1:ncpus=24,walltime=02:00:00

# change to local scratch directory
SCR=/scratch/work/user/$USER/$PBS_JOBID
mkdir -p $SCR ; cd $SCR || exit

# get individual tasks from tasklist with index from PBS JOB ARRAY
TASK=$(sed -n "${PBS_ARRAY_INDEX}p" $PBS_O_WORKDIR/tasklist)

# copy input file and executable to scratch
cp $PBS_O_WORKDIR/$TASK input ; cp $PBS_O_WORKDIR/myprog.x .

# execute the calculation
./myprog.x < input > output

# copy output file to submit directory
cp output $PBS_O_WORKDIR/$TASK.out
```

In this example, the submit directory holds the 900 input files, executable `myprog.x` and the jobscript file. As input for each run, we take the filename of input file from created tasklist file. We copy the input file to scratch `/scratch/work/user/USER/PBS_JOBID`, execute the `myprog.x` and copy the output file back to >the submit directory, under the `$TASK.out` name. The `myprog.x` runs on one node only and must use threads to run in parallel. Be aware, that if the `myprog.x` is **not multithreaded**, then all the **jobs are run as single thread programs in sequential** manner. Due to allocation of the

whole node, the **accounted time is equal to the usage of whole node**, while using only 1/24 of the node!

If huge number of parallel multicore (in means of multinode multithread, e. g. MPI enabled) jobs is needed to run, then a job array approach should also be used. The main difference compared to previous example using one node is that the local scratch should not be used (as it's not shared between nodes) and MPI or other technique for parallel multinode run has to be used properly.

### Submit the job array

To submit the job array, use the `qsub -J` command. The 900 jobs of the example above may be submitted like this:

```
$ qsub -N JOBNAME -J 1-900 jobscript
506493[].isrv5
```

In this example, we submit a job array of 900 subjobs. Each subjob will run on full node and is assumed to take less than 2 hours (please note the `#PBS` directives in the beginning of the jobscript file, don't forget to set your valid `PROJECT_ID` and desired queue).

Sometimes for testing purposes, you may need to submit only one-element array. This is not allowed by PBSPro, but there's a workaround:

```
$ qsub -N JOBNAME -J 9-10:2 jobscript
```

This will only choose the lower index (9 in this example) for submitting/running your job.

### Manage the job array

Check status of the job array by the `qstat` command.

```
$ qstat -a 506493[].isrv5
```

```
isrv5:
```

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Req'd Memory	Req'd Time	Elap S	Time
12345[].dm2	user2	qprod	xx	13516	1	24	--	00:50 B	00:02	

The status B means that some subjobs are already running.

Check status of the first 100 subjobs by the `qstat` command.

```
$ qstat -a 12345[1-100].isrv5
```

```
isrv5:
```

Job ID	Username	Queue	Jobname	SessID	NDS	Req'd TSK	Req'd Memory	Elap Time	S	Time
12345[1].isrv5	user2	qprod	xx	13516	1	24	--	00:50	R	00:02
12345[2].isrv5	user2	qprod	xx	13516	1	24	--	00:50	R	00:02
12345[3].isrv5	user2	qprod	xx	13516	1	24	--	00:50	R	00:01
12345[4].isrv5	user2	qprod	xx	13516	1	24	--	00:50	Q	--
.	.	.	.	.	.	.	.	.	.	.
,	.	.	.	.	.	.	.	.	.	.
12345[100].isrv5	user2	qprod	xx	13516	1	24	--	00:50	Q	--

Delete the entire job array. Running subjobs will be killed, queueing subjobs will be deleted.

```
$ qdel 12345[] .isrv5
```

Deleting large job arrays may take a while.

Display status information for all user's jobs, job arrays, and subjobs.

```
$ qstat -u $USER -t
```

Display status information for all user's subjobs.

```
$ qstat -u $USER -tJ
```

Read more on job arrays in the PBSPRO Users guide.

## GNU parallel

Use GNU parallel to run many single core tasks on one node.

GNU parallel is a shell tool for executing jobs in parallel using one or more computers. A job can be a single command or a small script that has to be run for each of the lines in the input. GNU parallel is most useful in running single core jobs via the queue system on Anselm.

For more information and examples see the parallel man page:

```
$ module add parallel
$ man parallel
```

## GNU parallel jobscript

The GNU parallel shell executes multiple instances of the jobscript using all cores on the node. The instances execute different work, controlled by the \$PARALLEL\_SEQ variable.

Example:

Assume we have 101 input files with name beginning with “file” (e. g. file001, ..., file101). Assume we would like to use each of these input files with program executable myprog.x, each as a separate single core job. We call these single core jobs tasks.

First, we create a tasklist file, listing all tasks - all input files in our example:

```
$ find . -name 'file*' > tasklist
```

Then we create jobscript:

```
#!/bin/bash
#PBS -A PROJECT_ID
#PBS -q qprod
#PBS -l select=1:ncpus=24,walltime=02:00:00

[ -z "$PARALLEL_SEQ" ] &&
{ module add parallel ; exec parallel -a $PBS_O_WORKDIR/tasklist $0 ; }

# change to local scratch directory
SCR=/scratch/work/user/$USER/$PBS_JOBID/$PARALLEL_SEQ
mkdir -p $SCR ; cd $SCR || exit

# get individual task from tasklist
TASK=$1

# copy input file and executable to scratch
cp $PBS_O_WORKDIR/$TASK input

# execute the calculation
cat input > output
```

```
# copy output file to submit directory
cp output $PBS_O_WORKDIR/$TASK.out
```

In this example, tasks from tasklist are executed via the GNU parallel. The jobscript executes multiple instances of itself in parallel, on all cores of the node. Once an instance of jobscript is finished, new instance starts until all entries in tasklist are processed. Currently processed entry of the joblist may be retrieved via \$1 variable. Variable \$TASK expands to one of the input filenames from tasklist. We copy the input file to local scratch, execute the myprog.x and copy the output file back to the submit directory, under the \$TASK.out name.

### Submit the job

To submit the job, use the qsub command. The 101 tasks’ job of the example above may be submitted like this:

```
$ qsub -N JOBNAME jobscript
12345.dm2
```

In this example, we submit a job of 101 tasks. 24 input files will be processed in parallel. The 101 tasks on 24 cores are assumed to complete in less than 2 hours.

Please note the #PBS directives in the beginning of the jobscript file, don't forget to set your valid PROJECT\_ID and desired queue.

## Job arrays and GNU parallel

Combine the Job arrays and GNU parallel for best throughput of single core jobs

While job arrays are able to utilize all available computational nodes, the GNU parallel can be used to efficiently run multiple single-core jobs on single node. The two approaches may be combined to utilize all available (current and future) resources to execute single core jobs.

Every subjob in an array runs GNU parallel to utilize all cores on the node

## GNU parallel, shared jobscript

Combined approach, very similar to job arrays, can be taken. Job array is submitted to the queuing system. The subjobs run GNU parallel. The GNU parallel shell executes multiple instances of the jobscript using all cores on the node. The instances execute different work, controlled by the \$PBS\_JOB\_ARRAY and \$PARALLEL\_SEQ variables.

Example:

Assume we have 992 input files with name beginning with "file" (e. g. file001, ..., file992). Assume we would like to use each of these input files with program executable myprog.x, each as a separate single core job. We call these single core jobs tasks.

First, we create a tasklist file, listing all tasks - all input files in our example:

```
$ find . -name 'file*' > tasklist
```

Next we create a file, controlling how many tasks will be executed in one subjob

```
$ seq 32 > numtasks
```

Then we create jobscript:

```
#!/bin/bash
#PBS -A PROJECT_ID
#PBS -q qprod
```

```

#PBS -l select=1:ncpus=24,walltime=02:00:00

[ -z "$PARALLEL_SEQ" ] &&
{ module add parallel ; exec parallel -a $PBS_O_WORKDIR/numtasks $0 ; }

# change to local scratch directory
SCR=/scratch/work/user/$USER/$PBS_JOBID/$PARALLEL_SEQ
mkdir -p $SCR ; cd $SCR || exit

# get individual task from tasklist with index from PBS JOB ARRAY and index form Parallel
IDX=$(( $PBS_ARRAY_INDEX + $PARALLEL_SEQ - 1 ))
TASK=$(sed -n "${IDX}p" $PBS_O_WORKDIR/tasklist)
[ -z "$TASK" ] && exit

# copy input file and executable to scratch
cp $PBS_O_WORKDIR/$TASK input

# execute the calculation
cat input > output

# copy output file to submit directory
cp output $PBS_O_WORKDIR/$TASK.out

```

In this example, the jobscript executes in multiple instances in parallel, on all cores of a computing node. Variable \$TASK expands to one of the input filenames from tasklist. We copy the input file to local scratch, execute the myprog.x and copy the output file back to the submit directory, under the \$TASK.out name. The numtasks file controls how many tasks will be run per subjob. Once an task is finished, new task starts, until the number of tasks in numtasks file is reached.

Select subjob walltime and number of tasks per subjob carefully

When deciding this values, think about following guiding rules :

1. Let  $n=N/24$ . Inequality  $(n+1) * T < W$  should hold. The N is number of tasks per subjob, T is expected single task walltime and W is subjob walltime. Short subjob walltime improves scheduling and job throughput.
2. Number of tasks should be modulo 24.
3. These rules are valid only when all tasks have similar task walltimes T.

### Submit the job array

To submit the job array, use the qsub -J command. The 992 tasks' job of the example above may be submitted like this:

```
$ qsub -N JOBNAME -J 1-992:32 jobscript
```

```
12345[] .dm2
```

In this example, we submit a job array of 31 subjobs. Note the `-J 1-992:48`, this must be the same as the number sent to numtasks file. Each subjob will run on full node and process 24 input files in parallel, 48 in total per subjob. Every subjob is assumed to complete in less than 2 hours.

Please note the `#PBS` directives in the beginning of the jobscript file, don't forget to set your valid `PROJECT_ID` and desired queue.

## Examples

Download the examples in `capacity.zip`, illustrating the above listed ways to run huge number of jobs. We recommend to try out the examples, before using this for running production jobs.

Unzip the archive in an empty directory on Anselm and follow the instructions in the README file

```
$ unzip capacity.zip
$ cd capacity
$ cat README
```