Example Workflows

On this page:

Job Arrays

An array job provides a mechanism to run a (possibly large) number of similar jobs from a single batch script. The number of instances of the job is controlled via the SLURM directive

```
#SBATCH --array=list
```

Here list may be a comma-separated list of indices, or a range of indices specified using a dash "-". For example, one may have "--array=0,1,2,3" or "--array=0-3" to specify four instances. A combination of these two formats may be used, e.g., "--array=0-2,4,8". An optional stride may be introduced when specifying a range using a colon ":", e.g., "--array=0-7:2", being equivalent to "--array=0,2,4,6".

All the other SLURM directives specified in the script are common to all the jobs, specifically the number of nodes, and the time limit.

The following simple example runs two instances of a 24 MPI task job, each on one node.

```
#!/bin/bash --login

# SLURM directives
#
# This is an array job with two subtasks 0 and 1 (--array=0,1).
#
# The output for each subtask will be sent to a separate file
# identified by the jobid (--output=array-%j.out)
#
# Each subtask will occupy one node (--nodes=1) with
# a wall-clock time limit of one minute (--time=00:01:00)
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123)

#SBATCH --array=0,1
#SBATCH --output=array-%j.out
#SBATCH --nodes=1
#SBATCH --ntasks=24                         #this directive is required on Zeus to request 24 tasks on one node
#SBATCH --time=00:01:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# To launch the job, we specify to srun 24 MPI tasks (-n 24)
# to run on the node
#
# Note we avoid any inadvertent OpenMP threading by setting
# OMP_NUM_THREADS=1
#
# The input to the execuatable is the unique array task identifier
# $SLURM_ARRAY_TASK_ID which will be either 0 or 1

export OMP_NUM_THREADS=1

echo This job shares a SLURM array job ID with the parent job: $SLURM_ARRAY_JOB_ID
echo This job has a SLURM job ID: $SLURM_JOBID
echo This job has a unique SLURM array index: $SLURM_ARRAY_TASK_ID

srun -N 1 -n 24 ./code_mpi.x $SLURM_ARRAY_TASK_ID
```

The job is submitted as normal:

```
sbatch array_script.sh
Submitted batch job 212681
```

The "parent" job will initially appear in the queue with an underscore appended to the jobid, e.g., "212681_". The first sub-job, when started, will appear with the same job id as the parent but without the underscore. Subsequent sub-jobs have consecutive job ids which in this case give output, e.g., "array-212681.out" and "array-212682.out".

Job Dependencies

It is possible to specify dependencies between two jobs using a unique SLURM job id. Suppose Job 2 cannot start until the successful completion of Job 1, but we want to submit Job 1 and Job 2 at the same time. So, submit Job 1 as usual:

```
sbatch job1-script.sh
Submitted batch job 206842
```

We can then immediately submit Job 2 specifying the dependency on Job 1 (id 206842) using the -d option:

```
sbatch -d afterok:206842 job2-script.sh
Submitted batch job 206845
```

At this point Job 2 will enter the queue in the pending state and appear under squeue as having a dependency. The clause afterok:id means Job 2 will not become eligible to run until Job 1 has finished successfully (that is, job1-script.sh exits with exit code zero). If Job 1 does exit successfully, Job 2 will become eligible to run and will run at the next opportunity. However, if Job 1 fails, Job 2 can never run, and will be (silently) removed from the queue.

A number of additional types of dependency are available. These include:

Option	Purpose
-d afterany:jobid	Dependent job may run after any exit status
-d afternotok:jobid	Dependent job may run only after non-zero exit status

A group of jobs can be submitted one after the other, utilizing the previous JOBID to create a dependency between the individual jobs.

The following example has four jobs that are dependent on the previously submitted job.

```
jobid=`sbatch first_job.sh | cut -d " " -f 4`
jobid=`sbatch --dependency=afterok:$jobid second_job.sh | cut -d " " -f 4`
jobid=`sbatch --dependency=afterok:$jobid third_job.sh | cut -d " " -f 4`
sbatch --dependency=afterok:$jobid fourth_job.sh
```

The more complicated example below has the first three jobs as being independent, but the last job only runs after the previous three have completed successfully.

```
jobid1=`sbatch first_job.sh | cut -d " " -f 4`
jobid2=`sbatch second_job.sh | cut -d " " -f 4`
jobid3=`sbatch third_job.sh | cut -d " " -f 4`
sbatch --dependency=afterok:$jobid1:$jobid2:$jobid3 fourth_job.sh
```

Another common method of creating dependencies between jobs is to have the batch job submit a new job at the end of the job script (job chaining).

```
#!/bin/bash -l
#SBATCH --account=[your-project]
#SBATCH --nodes=xx
#SBATCH --ntasks=yy                        #this directive is required on Zeus to request yy tasks
#SBATCH --time=00:05:00
#SBATCH --export=NONE

srun -N xx -n yy ./a.out # fill in the srun options '-N xx/-n yy/etc' to be appropriate to run the job
sbatch next_job.sh
```

Dependencies may be used within a SLURM script itself by making use of the SLURM variable $SLURM_JOB_ID to identify the current job. For example,

```
#!/bin/bash -l
#SBATCH --account=[your-project]
#SBATCH --nodes=xx
#SBATCH --ntasks=yy                        #this directive is required on Zeus to request yy tasks
#SBATCH --time=00:05:00
#SBATCH --export=NONE

sbatch --dependency=afternotok:${SLURM_JOB_ID} next_job.sh
srun -N xx -n yy ./code.x # fill in the srun options '-N xx -n yy' etc. to be appropriate to run the job
```

Job dependencies and job priorities

Please note that submitting multiple jobs and using dependencies will not obtain a higher queue priority for the dependent jobs just because they were submitted earlier. Accrual of job age priority starts from the eligible time, not the submission time. Jobs with dependencies only become eligible when the dependency is removed/completed.

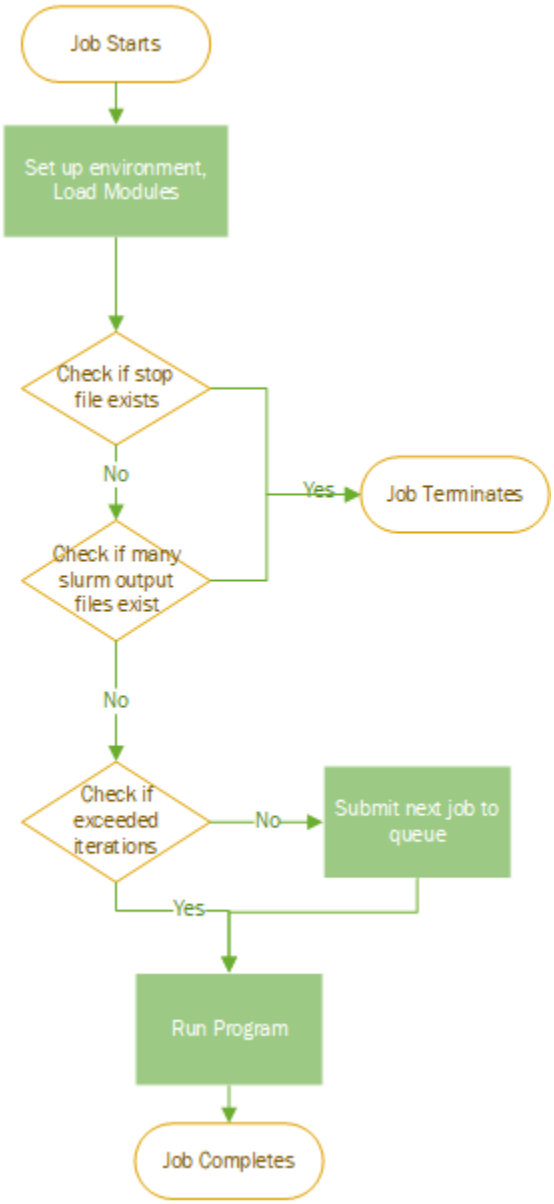
```
reaper@magnus-1:~> squeue -j 4979452,4979463,4979465 -O "jobid,submittime,eligibletime,reason,dependency"
JOBID               SUBMIT_TIME         ELIGIBLE_TIME       REASON              DEPENDENCY
4979452             2020-05-22T09:47:45 2020-05-22T09:47:45 Priority
4979463             2020-05-22T09:49:19 2020-05-22T10:20:54 Priority
4979465             2020-05-22T09:49:38 N/A                 Dependency          afternotok:4979463
```

In the above example, job 4979452 was submitted with no dependency, and became eligible to run immediately. Job 4979463 was submitted with a dependency, but that dependency finished at 10:20:54 so this job was now eligible to run. Job 4979465 is still not eligible to run.

Recursive Jobs

When a job cannot be completed within the walltime of 24 hours, it will need to be restarted from its last checkpoint. If the job needs to be restarted several times before reaching completion, it is convenient to allow subsequent jobs to restart automatically rather than submitting them manually. For this, the initial job script needs to contain the logic for submitting subsequent jobs automatically. Obviously, the code that is being executed needs to be able to restart from an existing checkpoint generated from the previous run. And, usually, it also needs an updated version of the restart parameters. Therefore, the initial job script also needs to be equipped with the necessary updating procedure to allow the use of the existing checkpoint file and the new input parameters.

The below image shows the logic flow of the following recursive job script example. Three check levels are included in the example. First check is for the existence of a file within the case directory which used utilised as an indicator for the process to terminate. The next check counts how many job-output-files have been generated, if the specified maximum has been reached then the job is terminated. And the third check examines the counter of how many job submissions (iterations) have been performed, if the specified maximum has been reached, then the rest of the current script will proceed until the end and finalise the whole process without submitting any new job.



From the three check levels suggested above, the third one is the most intuitive from a "programming" logic perspective. The first two have been added as additional "safety net" checks. The second one has been included to avoid the creation of an infinite loop of re-submissions when there is some bug in logic of the script. And the first check allows the user (or any other sub process/script) to raise a flag of termination when a dummy "flag-to-stop" file is created.

In the following paragraphs we'll explain the logic with an example.

First of all, if the main executable (`code.x`) in the job script needs to read its input parameters from a file (`input.dat` in this case), then the script may need to adapt the input parameters for each job execution. For dealing with that, let's assume as an example that the `input.dat` file is something like this:

input.dat

```
starttime=0
endtime=10
```

These parameters are used by `code.x` to define the inital and final times of the numerical simulation it executes. As the job will be submitted many times recursively, the file of input parameters needs to be updated at every recursion before running the executable. For that, we'll make use of a template file named "`input.template`":

input.template

```
starttime=VAR_START_TIME
endtime=VAR_END_TIME
```

This template file will replace the `input.dat` file and the strings `VAR_START_TIME` and `VAR_END_TIME` will be replaced by the needed values in each recursion (using the command "`sed`") before running the executable in the current job. This logic is in the section "`##Setup/Update of parameters /input for this current script`" of the example script presented a couple of paragraphs below.

For this process to work properly, the correct values of the input parameters need to be set and "sent" to the following job submission. This is performed when the submission of the following dependent job is done. This logic is in the section "`##Submitting the dependent job`" of the example script presented a couple of paragraphs below.

The example script "`iterative.sh`" performs the logic described above (comments within explain the reasoning of each section):

```
#!/bin/bash -l
#-----------------------
##Defining the needed resources with SLURM parameters (modify as needed)
#SBATCH --account=[your-project]
#SBATCH --job-name=iterativeJob
#SBATCH --ntasks=xx
#SBATCH --ntasks-per-node=yy
#SBATCH --time=00:05:00
#SBATCH --export=NONE

#-----------------------
##Setting modules
#Add the needed modules (uncomment and adapt the follwing lines)
#module swap the-module-to-swap the-module-i-need
#module load the-modules-i-need

#-----------------------
##Setting the variables for controlling recursion
#job iteration counter. It's default value is 1 (as for the first submission). For a subsequent submission, it
will receive it value through the "sbatch --export" command from the "parent job".
: ${job_iteration:="1"}
this_job_iteration=${job_iteration}

#Maximum number of job iterations. It is always good to have a reasonable number here
job_iteration_max=5

echo "This jobscript is calling itself in recursively. This is iteration=${this_job_iteration}."
echo "The maximum number of iterations is set to job_iteration_max=${job_iteration_max}."
echo "The slurm job id is: ${SLURM_JOB_ID}"

#-----------------------
##Defining the name of the dependent script.
#This "dependentScript" is the name of the next script to be executed in workflow logic. The most common and
more utilised is to re-submit the same script:
thisScript=`squeue -h -j $SLURM_JOBID -o %o`
export dependentScript=${thisScript}

#-----------------------
##Safety-net checks before proceding to the execution of this script

#Check 1: If the file with the exact name 'stopSlurmCycle' exists in the submission directory, then stop
execution.
#         Users can create a file with this name if they need to interrupt the submission cycle by using the
following command:
#             touch stopSlurmCycle
#         (Remember to remove the file before submitting this script again.)
if [[ -f stopSlurmCycle ]]; then
   echo "The file \"stopSlurmCycle\" exists, so the script \"${thisScript}\" will exit."
   echo "Remember to remove the file before submitting this script again, or the execution will be stopped."
   exit 1
fi
```

```
#Check 2: If the number of output files has reached a limit, then stop execution.
#         The existence of a large number of output files could be a sign of an infinite recursive loop.
#         In this case we check for the number of "slurm-XXXX.out" files.
#         (Remember to check your output files regularly and remove the not needed old ones or the execution
may be stoppped.)
maxSlurmies=25
slurmyBaseName="slurm" #Use the base name of the output file
slurmies=$(find . -maxdepth 1 -name "${slurmyBaseName}*" | wc -l)
if [ $slurmies -gt $maxSlurmies ]; then
   echo "There are slurmies=${slurmies} ${slurmyBaseName}-XXXX.out files in the directory."
   echo "The maximum allowed number of output files is maxSlurmies=${maxSlurmies}"
   echo "This could be a sign of an infinite loop of slurm resubmissions."
   echo "So the script ${thisScript} will exit."
   exit 2
fi

#Check 3: Add some other adequate checks to guarantee the correct execution of your workflow
#Check 4: etc.

#-----------------------
##Setup/Update of parameters/input for the current script

#The following variables will receive a value with the "sbatch --export" submission from the parent job.
#If this is the first time this script is called, then they will start with the default values given here:
: ${var_start_time:="0"}
: ${var_end_time:="10"}
: ${var_increment:="10"}

#Replacing the current values in the parameter/input file used by the executable:
paramFile=input.dat
templateFile=input.template
cp $templateFile $paramFile
sed -i "s,VAR_START_TIME,$var_start_time," $paramFile
sed -i "s,VAR_END_TIME,$var_end_time," $paramFile

#Creating the backup of the parameter file utilised in this job
cp $paramFile $paramFile.$SLURM_JOB_ID

#-----------------------
##Verify that everything that is needed is ready
#This section is IMPORTANT. For example, it can be used to verify that the results from the parent submission
are there. If not, stop execution.

#-----------------------
##Submitting the dependent job
#IMPORTANT: Never use cycles that could fall into infinite loops. Numbered cycles are the best option.

#The following variable needs to be "true" for the cycle to proceed (it can be set to false to avoid recursion
when testing):
useDependentCycle=true

#Check if the current iteration is within the limits of the maximum number of iterations, then submit the
dependent job:
if [ "$useDependentCycle" = "true" ] && [ ${job_iteration} -lt ${job_iteration_max} ]; then
   #Update the counter of cycle iterations
   (( job_iteration++ ))
   #Update the values needed for the next submission
   var_start_time=$var_end_time
   (( var_end_time += $var_increment ))
   #Dependent Job submission:
   #                         (Note that next_jobid has the ID given by the sbatch)
   #                         For the correct "--dependency" flag:
   #                         "afterok", when each job is expected to properly finish.
   #                         "afterany", when each job is expected to reach walltime.
   #                         "singleton", similar to afterany, when all jobs will have the same name
   #                         Check documentation for other available dependency flags.
   #IMPORTANT: The --export="list_of_exported_vars" guarantees that values are inherited to the dependent job
   next_jobid=$(sbatch --export="job_iteration=${job_iteration},var_start_time=${var_start_time},
var_end_time=${var_end_time},var_increment=${var_increment}" --dependency=afterok:${SLURM_JOB_ID}
${dependentScript} | awk '{print $4}')
   echo "Dependent with slurm job id ${next_jobid} was submitted"
```

```
   echo "If you want to stop the submission chain it is recommended to use scancel on the dependent job first"
   echo "Or create a file named: \"stopSlurmCycle\""
   echo "And then you can scancel this job if needed too"
else
   echo "This is the last iteration of the cycle, no more dependent jobs will be submitted"
fi

#-----------------------
##Run the main executable.
#(Modify as needed)
#Syntax should allow restart from a checkpoint
srun -N $SLURM_JOB_NUM_NODES -n $SLURM_NTASKS ./code.x
```

Users can adapt this script to their needs, with special attention to the security checks, type of dependency needed and appropriate syntax for restarting from previous checkpoint.

In order to submit the initial job, this is performed in the usual way:

```
sbatch iterative.sh
```

In the first iteration, the job will run with the default value of `job_iteration=1,` and of the input parameters (`var_start_time=0, var_start_time=10, var_increment=10`) and use them to create the `input.dat` file for the current run. Before submitting the second job, those values will be updated. Then, they will be "sent through" in the `sbatch` submission command of the dependent job. The updated values of those variables and parameters will be received in the second job and will be utilised instead of the defaults.

Note that the `sbatch` submission uses:

```
--dependency=afterok:${SLURM_JOB_ID}
```

This means that the dependent job will be submitted to the queue, but Slurm will still wait for the parent job to finish properly. And only if the parent job did finish properly (`afterok`) the dependent job will be kept in the queue and continue its process. Other common dependency option is "`afterany`", which is of common use if the job is expected to reach the walltime in each submission. For other dependency options check Job Dependencies above.

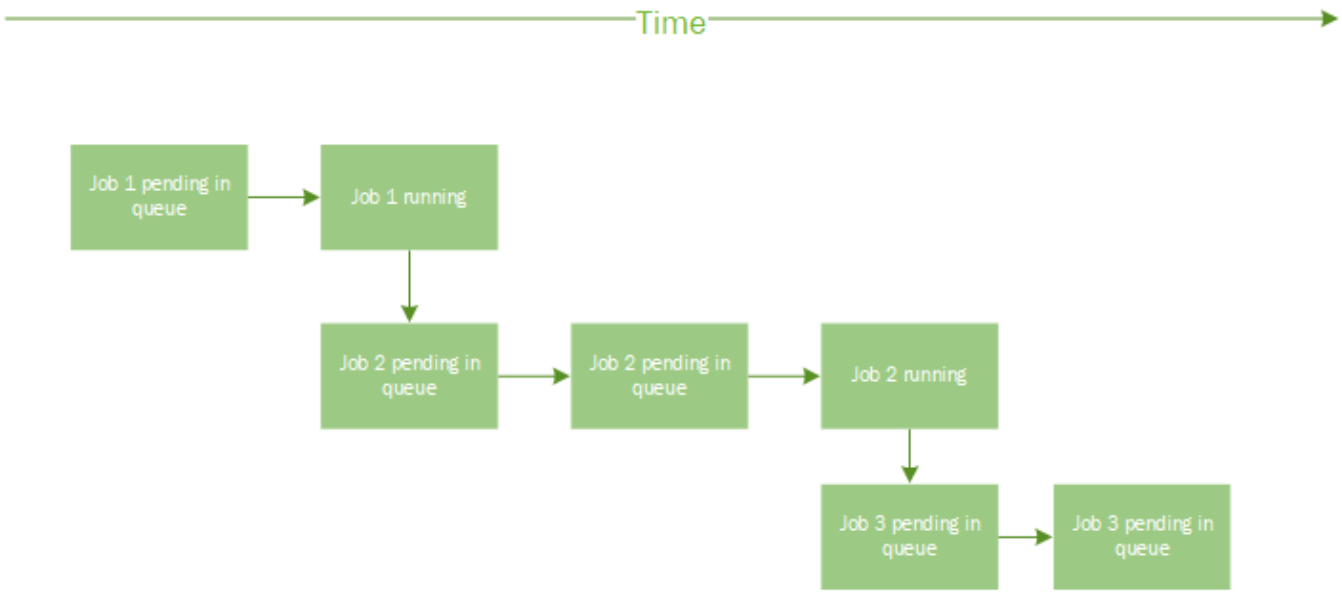
When a job is submitted with recursive capabilities, the `squeue` command may show a running job and a job waiting to be processed due to dependency. The dependent job will not be eligible to start until the running job has finished. As mentioned above in the Example Workflows#Jobdependenciesandjobpriorities section, the dependent job will not accrue age priority until the first job has completed.

```
espinosa@magnus-2:> squeue -u espinosa
JOBID    USER     ACCOUNT                  NAME EXEC_HOST ST     REASON   START_TIME     END_TIME  TIME_LEFT
NODES   PRIORITY
3483798  espinosa pawsey0001        iterativeJob  nid00017  R       None     14:44:27     14:47:27       2:
50     1       5269
3483799  espinosa pawsey0001        iterativeJob       n/a PD Dependency          N/A          N/A       5:
00     1       5269
```

Alternatively, if the running job has completed and the dependent job has not yet started, then you will only see the dependent job in the `squeue` output, and the REASON will be either Priority or Resources.

The below diagram summarises what to expect when looking at the job queue for this recursive job script.

- On initial job submission, there will be one job in the queue
- When a job is running, you will see another job appear in the queue that is held with REASON=Dependency
- When each iteration of the job has finished, there will again only be one pending job in the queue



If for some reason you want to stop the recursive submission, but without cancelling the current running job, you can create a file named "`stopSlurmCycle`" (check the example script above in the section "`##Safety-net checks`") with:

```
touch stopSlurmCycle
```

Or you can cancel the dependent job with:

```
scancel 3483799
```

The jobID of the dependent job was taken from the display of the `squeue` command above.

And, to cancel the whole process, you should cancel both jobs, the dependent and the running one. (It is always wise to cancel first the dependent job.)

Multi-cluster Jobs

SLURM queueing system offers the ability to launch commands on other clusters instead of, or in addition to, the local cluster on which the command is invoked. A classic example of data staging is presented in Data workflows section, which runs simulation in the Magnus work queue and upon completion launches a data copying job to the copy queue on the Zeus cluster.

Interactive Jobs

For code development, debugging and light-weight visualisation purposes, it is sometimes convenient to run on the back-end "interactively". This can be done using the SLURM command salloc. For example, from the front-end we can enter salloc to ask for one node to be allocated in the debugq partition:

```
magnus-1:~> salloc -p debugq --nodes=1
salloc: Pending job allocation 206121
salloc: job 206121 queued and waiting for resources
salloc: Granted job allocation 206121
magnus@nid00200:~>
```

While interactive access to the workq partition is available via salloc, interactive jobs do not get additional priority. This may mean long wait times for interactive requests to be satisfied if the machine is busy.

Note the change in prompt, which indicates you are now logged into the compute node (nid00200 in this case).

--ntasks option should also be used on Zeus to explicitly specify the number of tasks required for the interactive session.

You must use srun to run multiple instances of your executable in parallel.

For example:

```
magnus@nid00200:~> cd $MYGROUP
magnus@nid00200:/group/[project]/[username]>
magnus@nid00200:/group/[project]/[username]> srun -N 1 -n 4 ./code_test.x
...
```

When finished, type exit to leave the interactive queue and rejoin the front-end.

```
magnus@nid00200:~> exit
exit
salloc: Relinquishing job allocation 206121
salloc: Job allocation 206121 has been revoked
magnus-1:~>
```

Note that X11 forwarding is enabled by default in the interactive queue.

We recommend users to use FastX, a web-based remote visualisation service on Topaz to launch any compute-intensive visualisation packages, such as ParaView, VisIt, VMD etc., Please refer to the Remote Visualisation support page for more information.

Packing Serial/Small Multithreaded Jobs

Exploiting parallelism for a given workflow sometimes means running many copies of a serial code with different input parameters or data. Outputs must be stored separately and in an identifiable way. The same is true if we consider jobs which support threads, but do not scale to a full node. In this case we might want to run, say, 6 jobs each of 4 threads at the same time. For purposes of efficiency, we would like to pack a number of such instances in one node to make use of all cores available within the node (e.g. 24 on Magnus).

There are a number of ways in which this can be done:

1. For "trivial" parallelism, where all the tasks are completely independent, individual tasks can be uniquely identified by the environment variable SLURM_PROCID which takes on a value between 0 ... ntasks-1 when an application is launched srun -n ntasks. Examples are given below.
2. For more complex workflows, where there may be some dependencies between tasks, we recommend considering mpibash. See the section Using mpibash for more details.
3. For complex scripting tasks requiring parallelism, we suggest considering python and message passing via mpi4py. See the section Using Python and mpi4py for more information.

Method 1: Using SLURM_PROCID

Packing Serial Jobs

This section shows how to pack a work flow consisting of multiple serial (single core) instances of work on Magnus. The individual "instances of work" here might represent a serial binary executable or a separate serial script. In the example below, we use the environment variable SLURM_PROCID to identify input files and output files for each of 48 requested instances of a serial executable serial-code.x. Instead of launching the executable directly, an intermediate (wrapper) shell script is launched by srun. Inside the wrapper script one has access to $SLURM_PROCID and can construct the serial workflow that is intended to execute a given instance:

```
#!/bin/bash --login

# SLURM directives
#
# Here we specify to SLURM we want two node (--nodes=2) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=2
#SBATCH --ntasks=48                         #this directive is  required on Zeus to request 48 tasks
#SBATCH --ntasks-per-node=24         #this directive is required on Zeus to request 24 tasks on each node
#SBATCH --time=00:10:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# Launch the job.

# Launch 48 instances of wrapper script (make sure it's executable),
# with 24 on each node

srun -N 2 -n 48 ./wrapper.sh
```

The wrapper.sh may look something like this:

```
#!/bin/bash
#
# This is a standard bash script which has access to the environment variable SLURM_PROCID
# This is used to construct input filenames of the form input-0 input-1 ... input-47
# and similarly named output files

INFILE="input-$SLURM_PROCID"
OUTFILE="output-$SLURM_PROCID.out"

# Assuming all the input files exist in the current directory, we run the executable.
# Each instance will use the appropriate input and produce the relevant output.

./serial-code.x < $INFILE > $OUTFILE
```

Java Jobs

Example 1: Serial Java application

Here, we run a serial Java application (class Application) on one node.

```
#!/bin/bash --login

# SLURM directives
#
# Here we specify to SLURM we want one node (--nodes=1) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=1
#SBATCH --ntasks=1                                #this directive is required on Zeus to request 1 task
#SBATCH --time=00:10:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# Launch the job.
# There is one task to run java in serial (-n 1).

srun -n 1 java Application
```

Example 2: Two Java instances on one node

Running a single Java application on one node will not make use of all 24 cores on the node (although it might require the entire 64 GB available). If one wants to run a number of instances of an application on the same node, an intermediate (or wrapper) application must be used via srun. The following example uses two instances, which are identified via the environment variable SLURM_PROCID. This variable takes on a unique value (starting at zero) for each instance specified to srun.

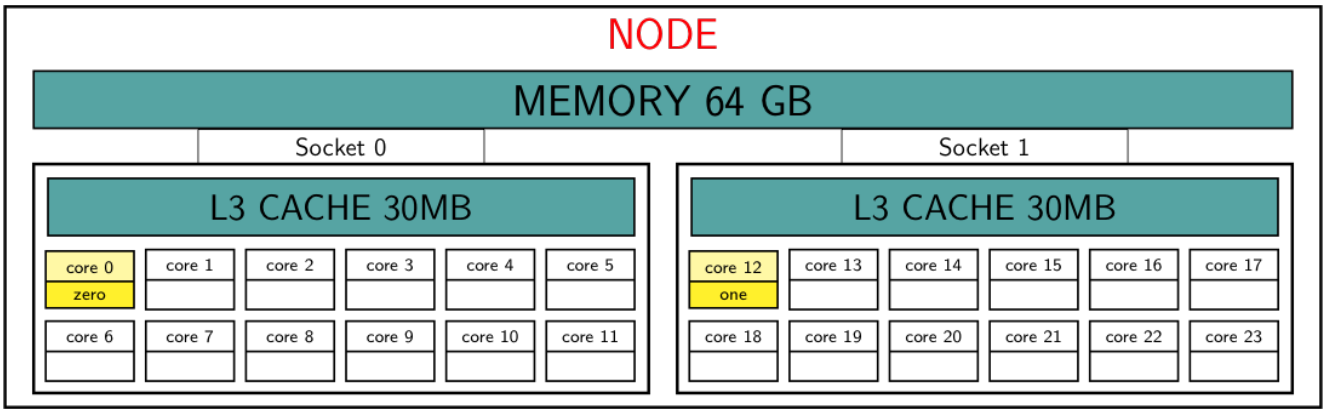
The SLURM script is as follows:

```
#!/bin/bash --login

# Here we specify to SLURM we want one node (--nodes=1) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=1
#SBATCH --ntasks=2                                #this directive is required on Zeus to request 2 tasks
#SBATCH --time=00:10:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# We request two instances "-n 2" to be placed on cores 0 and 12 "--cpu_bind=map_cpu:0,12"

srun -n 2 --cpu_bind=map_cpu:0,12 java Wrapper
```

The Wrapper.java application takes the form. Two instances of the Wrapper class are run (asynchronously)) which will be identical except for the value of SLURM_PROCID obtained from the environment. Appropriate program logic may be used to arrange, e.g., specific input to an instance of an underlying application. Here, we simply report the value of SLURM_PROCID to standard output.

```
/* Wrapper to differentiate instances produced by srun */
/* The resulting "rank" may be used in conjunction with
* program logic to run different tasks from within java. */
import java.io.*;
class Wrapper {
        public static void main(String argv[]) {
                int rank;
                try {
                        String slurm_proc_id = System.getenv("SLURM_PROCID");
                        rank = Integer.parseInt(slurm_proc_id);
                }
                catch (NumberFormatException e) {
                        rank = -1;
                }
                if (rank == 0) {
                        System.out.println("Running with SLURM_PROCID zero");
                }
                if (rank == 1) {
                        System.out.println("Running with SLURM_PROCID one");
                }
                /* ...and so on */
                return;
        }
}
```

The placement of the two instances, which are arranged to promote uniform access to half the memory (and half the memory bandwidth of the node) each on Magnus, is illustrated in the following diagram:



This is a general mechanism which may be used to run up to 24 instances on the same node, and may be used for tasks which are trivially parallel.

R Jobs

Interactive access to R for testing and development is available via the queue system. For interactive use

```
magnus-1:~> salloc --nodes=1 --time=06:00:00
salloc: Granted job allocation 291021
magnus@nid00294:~> module load cray-R
magnus@nid00294:~> srun -n 1 R --no-save

R version 3.3.3 (2017-03-06) -- "Another Canoe"
Copyright (C) 2017 The R Foundation for Statistical Computing
...
```


- The srun is required to ensure the R executable runs on the back end.
- The default time limit for the interactive queue is one hour (at the end of which you will be logged out automatically and without warning). Please be sure to specify a time limit which is long enough to complete the task at hand.
- For short jobs of up to one hour, you can use salloc -p debugq if the default workq is busy.

Trivial parallelism may be introduced by the following mechanism. An intermediate "wrapper" script is required between the SLURM submission script and the R script itself. A simple example is:

```
#!/bin/bash --login

# SLURM script requesting one node with a time limit of 20 minutes.
# Replace [your-project] with the appropriate project budget.

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24                #this is required on Zeus to request 24 tasks on a node
#SBATCH --time=00:20:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

module load cray-R #for magnus

module load r #on Zeus

# Launch 24 instances (-n 24) on 24 cores of the wrapper script
srun -N 1 -n 24 ./r-wrapper.sh
```

The wrapper script is r-wrapper.sh which must be in the same location as the submission script, and must be executable (chmod 740 r-wrapper.sh):

```
#!/bin/bash
#
# This script, running on the back-end, has access to the
# environment variable $SLURM_PROCID, which will take on
# values 0-23 when launched via srun -n 24.
#
# This is used as input to the R script, and to differentiate the output
# as r-job-<jobid>-<instance>.out (where the jobid is the same for each
# separate batch submission $SLURM_JOBID)

R --no-save "--args $SLURM_PROCID " < my-script.R > r-$SLURM_JOBID-$SLURM_PROCID.log
```

Finally, The R script (my-script.R) can be based on:

```
# The R script identifies its "rank" via the command line argument

args <- commandArgs(TRUE)

print ("This R script instance has input ")
print (args)
```

Packing Small Multithreaded Jobs

If your application supports multithreading, you may request via the -c option to srun the number of threads per instance on a node. We may request a number of instances (as long as the number of threads times the number of instances does not exceed the total number of cores available within a node). Here is an example using OpenMP on Magnus (a similar approach can be used for pthreaded applications).

```
#!/bin/bash --login

# SLURM directives
#
# Here we specify to SLURM we want two nodes (--nodes=2) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2         #this directive is required on Zeus to request 2 tasks on each node
#SBATCH --ntasks=4                              #this directive is required on Zeus to request a total of 4
tasks
#SBATCH --cpus-per-task=12                #this directive is required on Zeus to request 12 CPU cores for each
task
#SBATCH --time=00:10:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# Set number of threads
export OMP_NUM_THREADS=12

# Launch the job.
# Here we use 4 instances (-n 4) with 2 per node with one instance per socket,
# or NUMA region. Each instance requests 12 cores -c 12 (via -c $OMP_NUM_THREADS)
srun -N 2 -n 4 --cpu_bind=sockets -c ${OMP_NUM_THREADS} ./wrapper.sh
```

The wrapper script in this case will invoke an OpenMP code. Again, the wrapper script must use SLURM_PROCID to differentiate the individual tasks (here 0-3) in an appropriate way for the workflow. For pthreaded applications, the number of threads must be communicated to the wrapper script and must be consistent with the value specified by the "-c" option. (Note that in the case above, the number of threads is available to the wrapper script via the exported environment variable OMP_NUM_THREADS.)

Method 2: Using mpibash

An MPI implementation exists for bash, and is available via the module system. This provides an implementation of a limited number of key MPI routines and is described here. Using mpibash presents a simple way to parallelise workflows based on standard bash scripts.

A simple example is given in the following snippet. Programmers who have used MPI should be immediately familiar with the idiom:

```
#!/usr/bin/env mpibash

# Note the mpibash shebang
#
# The following command informs bash of the location of the mpi_init commnd
# which can then be used to initialise MPI

enable -f mpibash.so mpi_init

mpi_init

mpi_comm_rank rank
mpi_comm_size size

echo "Hello from bash mpi rank $rank of $size"

mpi_finalize
```

Don't forget to change the file permissions on the script so that anyone can execute it. Eg.

```
chmod a+x mpi-bash.sh
```

assuming you called the script "mpi-bash.sh". The script can be launched with the following SLURM script on Magnus:

```
#!/bin/bash --login

# We must load the mpibash module
#
# This particular example uses 48 MPI tasks on 2 nodes
#
# Note --export=none is necessary to avoid error messages of the form:
# _pmi_inet_listen_socket_setup:socket setup failed

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=24        #this directive is required on Zeus to request 24 tasks on each node
#SBATCH --ntasks=48                         #this directive is required on Zeus to request a total of 48 tasks
#SBATCH --time=00:02:00
#SBATCH --account=[your-project]
#SBATCH --export=none

module swap PrgEnv-cray PrgEnv-gnu         #this is required for Magnus
module load mpibash
export PMI_NO_PREINITIALIZE=1                 #this is required for Magnus
export PMI_NO_FORK=1                                 #this is required for Magnus
srun -N 2 -n 48 ./mpi-bash.sh
```

The bash script may contain anything appropriate for a normal workflow. It may not, however, attempt to launch a stand-alone MPI executable.

Method 3: Using Python and mpi4py

This example runs a single serial python script on a single node.

```
#!/bin/bash --login

# SLURM directives
#
# Here we specify to SLURM we want one node (--nodes=1) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=1
#SBATCH --ntasks=1                 #this directive is required on Zeus to request 1 task
#SBATCH --time=00:10:00
#SBATCH --account=[your-project]
#SBATCH --export=NONE

# Launch the job.
#
# Serial python script. Load the default python module with
#
# module load python
#
# Launch the script on the back end with srun -n 1

module load python
srun -n 1 python ./serial-python.py

#
# If you have an executable python script with a "bang path",
# make sure the path is of the form
#
# #!/usr/bin/env python

srun -n 1 ./serial-python-x.py
```

Suggestions on how to pack many serial tasks on a single node using mpi4py are given below.

```
#!/bin/bash --login

# SLURM directives
#
# Here we specify to SLURM we want two nodes (--nodes=2) with
# a wall-clock time limit of ten minutes (--time=00:10:00).
#
# Replace [your-project] with the appropriate project name
# following --account (e.g., --account=project123).

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=24        #this directive is required on Zeus to request 24 tasks on each node
#SBATCH --ntasks=48                        #this directive is required on Zeus to request a total of 48 tasks
#SBATCH --time=00:10:00
#SBATCH --account=[your-account]
#SBATCH --export=NONE

# Launch the job.
# This python script uses the python module mpi4py, which we need
# to load with
#
# module load mpi4py
#
# (which will also load the default python module as a dependency).
#
# The script is launched via srun -n 48, which specifies 48 MPI
# tasks, and is invoked via the interpreter.

module load mpi4py
srun -N 2 -n 48 python ./mpi-python.py

#
# If you have an executable python script with a "bang path",
# make sure the path is of the form
#
# #!/usr/bin/env python

srun -N 2 -n 48 ./mpi-python-x.py
```