Dakota User's Guide on ARCHER

Gordon Gibb, EPCC

Version 1.0, March 22, 2017

Contents

1. Introduction to Dakota

Dakota[1] is a toolkit that automates running a series of simulations whose input parameters can be varied in order to determine their effects on the simulation results. In particular, Dakota can be used to determine optimal parameter values, or quantify a model's sensitivity to varying parameters. The following classes of algorithm are currently supported by Dakota:

- Parameter Studies - For a list of parameter values to explore, Dakota will run a series of simulations using these given parameters and will tabulate the results.
- Design Of Experiments - Similar to parameter studies, but designed to achieve good coverage of the parameter space with minimal simulations.

- Uncertainty Quantification - Given an uncertainty on an input parameter (or several input parameters), Dakota will quantify the uncertainty of the simulation outputs.
- Optimisation - Dakota can find the best parameter/set of parameters to optimise a response function from a simulation.
- Calibration - Attempt to maximise agreement between simulation outputs and experimental data.

For more details, please refer to the Dakota User's Manual[4].

Virtually any simulation can be interfaced with Dakota, which treats a simulation code as a 'black box' that it feeds inputs (parameter values) into and receives outputs (response functions) from. The simplest way to interface a simulation with Dakota is through passing files between the simulation code and Dakota. In this case, Dakota will produce a parameter file and launch a user-written script that reads in the parameter file, parses it, and produces simulation input file(s). The script then launches the simulation, and once the simulation has completed, runs some post-processing on the simulation's results, returning the results to Dakota via a results file. Figure 1 outlines this approach. It is also possible to directly interface a simulation code with Dakota, however this requires knowledge of the Dakota source code, and is beyond the scope of this document. Interested readers can find out more about this in the Dakota Developer's Manual[2]

For some algorithms, the number of simulations that must be carried out and sets of parameters that must be investigated are known before Dakota is run. An example of this would be a parameter study, where the researcher knows which sets of parameters they wish to investigate. In these cases, Dakota merely automates the process of setting up, running, and tabulating the results of the simulations. Dakota's real power lies in cases where the number of simulations required, or the parameter values to be tested are not known beforehand. In this case, Dakota can automatically choose the parameters to be used for subsequent simulations (based upon whichever algorithm is being employed), and runs them. An example of this would be an optimisation algorithm, where the choice of parameters for the $n+1^{\text{th}}$ simulation may be dependent on the results of the n^{th} simulation. This Dakota functionality removes the need for a researcher to manually choose the

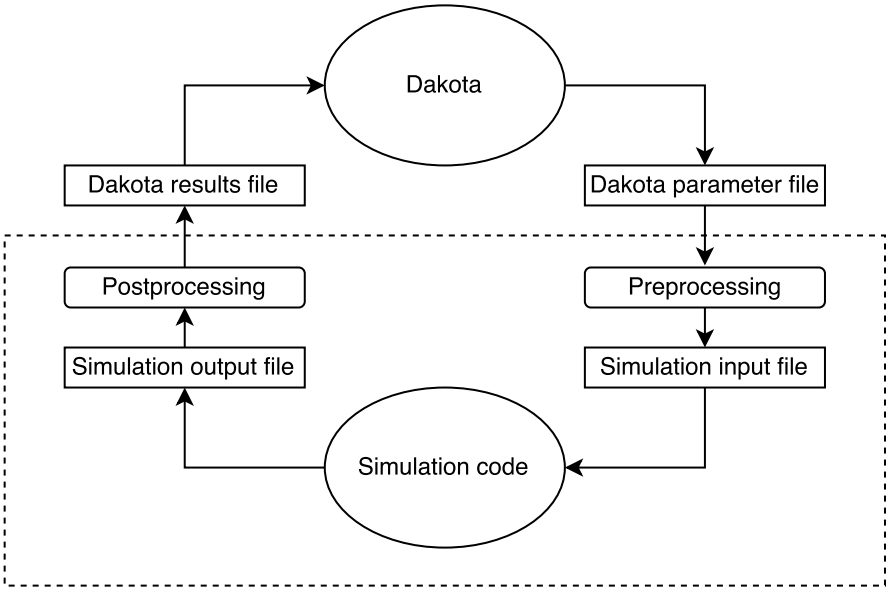

Figure 1: Flowchart describing how Dakota interfaces with a simulation. Dakota produces a parameter file, which is pre-processed into a simulation input file. The simulation code is then run, and its output file is then post-processed into a results file that is read in by Dakota, which can then start a new simulation with a new set of parameters if necessary. The portion of the flowchart contained within the dashed-line box is usually contained within an interface script, which handles preprocessing, simulation launching, and postprocessing.

parameters for subsequent simulations, prepare the input files, and run the simulations – saving them much time and effort.

This document is not intended to provide the reader with a complete understanding of all of Dakota's functionality and use cases, but rather to provide motivation for why Dakota may be applicable to their work, and how to set up their simulations and Dakota runs to work on ARCHER. For full details on Dakota and its functionality, please refer to the Documentation page on the Dakota website. A quick guide to use Dakota, and in particular the various use cases of Dakota on ARCHER will be outlined in Section 2.. A series of example scripts will be provided in Section 3. to cover the use cases outlined in Section 2.. Finally, some recommendations on how to get the most out of Dakota on ARCHER will be given in Section 4.

2. Using Dakota

2.1. Running Dakota

In order to run Dakota, it must have an input file which specifies the problem that Dakota is to solve and how to run a simulation. Optionally, an output file can be specified into which Dakota puts its output, and an error file into which any error messages go. If these files are not specified, dakota will send its output and error to standard output and error respectively. The run syntax for Dakota is:

```
dakota -i [input file] -o [output file] -e [error file]
```

When Dakota runs, it produces a restart file ('dakota.rst' by default) so that a dakota run can be restarted in the event of it being interrupted halfway through. In order to restart Dakota, use

```
dakota -i [input] -o [output] -e [error] -read_restart dakota.rst
```

On ARCHER, Dakota is available as a module. To access Dakota, use

```
module load dakota
```

2.2. Dakota Input Files

This subsection will describe the basic layout of an input file, however for full details on writing an input file, please read the Dakota User's Manual[4]. Dakota's input files are plain text files, with instructions put into a number of blocks. Some frequently used blocks are entitled `environment`, `method`, `variables`, `interface` and `responses`. Comments in the file are supported, and are initiated with a `#`. White space/indentation is not required, however it is useful for readability.

The `environment` block describes the general settings for Dakota, such as whether it should produce any additional output (such as tabulated values of the simulation parameters and response functions). The `method` block specifies the kind of study Dakota is to carry out, and specifies any options related to that study. The `variables` block specifies the variables in the simulation, such as parameters that Dakota can vary. The `interface` block describes how Dakota interfaces with a simulation code. Finally, the `responses` block specifies the response functions that the simulation returns to Dakota. There are several other blocks that can be used, however these are only necessary for certain kinds of study, and information about them can be found in the Dakota User's Manual[4]. An example Dakota input script is displayed in Figure 2.

2.3. Interfacing a Simulation with Dakota Using Scripts

As mentioned in Section 1., the simplest way to interface a simulation with Dakota is through the use of a script. This automates taking the simulation parameters from Dakota, producing a simulation input/configuration file, running the simulation, extracting the simulation's results and returning them to Dakota. Such an interface is shown in the example Dakota input file in Figure 2, and is known as a 'fork' interface. When using this interface, information is passed between Dakota and the script via two files: a parameter file and a results file. The names of these files are provided to the script as arguments.

Dakota comes with a tool called `dprepro` which parses the parameter file produced by Dakota and creates an input file for the simulation. In order to do this, a template file must be provided, which specifies the layout of the simulation's input file. In the template

```
# An example Dakota input file

environment
  tabular_data

method
  multidim_parameter_study
    partitions = 11 13

variables
  continuous_design = 2
    lower_bounds   0.    -2.5
    upper_bounds   1.     3.25
    descriptors    'a'    'b'

interface
  fork
    analysis_driver='script.sh'
    parameters_file='params.in'
    results_file='results.out'

responses
  objective_functions = 3
  no_gradients
  no_hessians
```

Figure 2: An example Dakota input file. In this example, Dakota is asked to tabulate its results of a multi-dimensional parameter study. The study investigates two variables, named 'a' and 'b', whose values range between $[0, 1]$ and $[-2.5, 3.25]$ respectively. The study will look at twelve values of 'a', and fourteen values of 'b'. Dakota will run the simulation using a system call (fork) to the script 'script.sh'. Dakota will pass parameter values into this script via the file 'params.in', and the script will pass simulation results into Dakota via the file 'results.out'. Finally, the simulation is expected to produce three objective functions, none of which are gradients or Hessians.

file, where the parameter value is to be inserted, the value is replaced by the name of the parameter (as specified in the Dakota input file) in curly braces. For example, if the simulation's parameter/configuration file takes three parameters as an input, 'a', 'b' and 'c', then the template file may look like:

```
a = {a}
b = {b}
c = {c}
```

and `dprepro` will automatically replace the items inside the curly braces with the values of 'a', 'b' and 'c' from Dakota's parameter file. The syntax for `dprepro` is:

```
dprepro [Dakota param file] [template] [simulation param/config file]
```

Once the simulation has finished, the script should apply some post-processing to extract the response functions from the simulation's results and produce the results file which is read by Dakota. This results file should contain the response functions separated by spaces. For example, if the simulation produces two response functions, equal to 3.14159 and 2.71828, then the Dakota results file should be:

```
3.14159   2.71828
```

2.4. ARCHER Use Cases

As mentioned in Section 1., the power in Dakota lies in its ability to automatically run simulations in order to carry out a study. Dakota is also capable of running multiple concurrent simulations, which on a large parallel system such as ARCHER can speed up carrying out such a study. Dakota may be used on the login nodes, the MOM nodes, or on compute nodes. The kind of node it is run on depends on the use case. On ARCHER there are four main parallel use cases for Dakota:

1. Run many concurrent single/few-core (shared-memory) simulations.
2. Run many concurrent short (on the order of a few hours) multi-processor/node simulations.

3. Run many consecutive short (on the order of a few hours) multi-processor/node simulations.

4. Run consecutive large (uses many nodes and/or runs for a very long time) multi-node simulations, or run many simulations that each require different core-counts (e.g. a speedup analysis).

The Dakota executable supports being run in parallel, and so can be run on the compute nodes via `aprun`. Every Dakota process can be used to launch a serial or a shared-memory (OpenMP or Pthread) simulation. This is the approach that should be used in Use Case 1. When running on the MOM nodes, dakota can launch MPI simulations using `aprun`. To do this, the user would submit a job using Dakota, requesting a number of nodes. Dakota will then run on the MOM node, and launch simulations as necessary using an `aprun` command in the interface scripts. In such a way, it can either run multiple concurrent multi-process/node simulations, each using a fraction of the nodes requested (Use Case 2), or it can run consecutive multi-process/node simulations using all of the nodes requested (Use Case 3). On a login node, Dakota can only submit jobs via `qsub`, which is placed inside the interface script. In such a case, Dakota should be called twice; once to get it to provide parameters to the simulation and submit a job to the queue, and a second time once the simulation has finished to return the results to Dakota. This approach (Use Case 4) should be used if each simulation that is to be carried out by Dakota takes a very long time to run (approaching 24h) so that it would be impractical to use Use Case 3 as only one simulation would be able to run per job. Additionally, if a study requires running simulations with different core counts then Use Case 4 is most suitable, since applying Use Cases 2 and 3 to such a study could waste compute time due to the nodes reserved for the study being under-utilised during some runs. Examples of each use case (and the corresponding files) will be given in Section 3..

3. Examples

In this Section example scripts, Dakota input files and PBS batch scripts will be given for each use case from Section 2.4.. They will also cover a number of different algorithms that Dakota has to offer.

3.1. Use Case 1: Multidimensional Parameter Study

Let us assume that we have a simulation code that runs on only a few cores at a time (using shared-memory parallelism), and only takes a few minutes to an hour to run. We wish to investigate how certain output values of the code vary according to a number of input parameters. In such a case, Use Case 1 is well suited to the problem, where we can run many small simulations in parallel.

As Dakota is launched using MPI in Use Case 1, the simulation code cannot use MPI, since nested MPI environments are not supported. Furthermore, in order to avoid instability the executable should be compiled without linking to ARCHER's MPI libraries. In order to do this, use:

```
module swap craype-network-aries craype-network-none
```

then compile your code using the normal `cc`, `CC` or `ftn` compiler wrappers. It is normal to get the warning message:

```
   Warning:
libraries from PE_MPICH will be ignored because they are not
compatible with network-target=none.
```

when compiling. This can be safely ignored. **Please note: the above steps are only necessary for Use Case 1, as the simulation executable cannot use MPI. For the other use cases, no special actions are needed to compile the simulation executable.**

Assume that we are interested in four input parameters, a, b, c and d, where a and b are continuous real numbers, whilst c and d are integer values. For each parameter, we wish

to investigate ten values, resulting in a total of 10,000 simulations needed to sample all the parameter values. The real variables both vary between 0 and 1, whilst c takes on values of 2^n, and d takes the values 10, 15, 25, ..., 55. There are three output values/response functions of interest, x, y and z. Let us assume that each simulation requires six threads to run optimally. We choose to run Dakota on 100 nodes, with 4 Dakota processes per node, resulting in 400 concurrent simulations.

There are three files we need to write: a Dakota input file, a script to interface the code with Dakota, and a PBS script to launch the job. The PBS script has to launch Dakota on the compute nodes using `aprun`. Figure 3 shows the Dakota input file, Figure 4 shows the interface script, and Figure 5 shows the PBS script.

```
environment
  tabular_data

method
  multidim_parameter_study
    #10 values of each parameter hence 9 partitions
    partitions = 9 9 9 9

variables
  continuous_design = 2
    lower_bounds  0.   0.
    upper_bounds  1.   1.
    descriptors  'a'   'b'
  discrete_design_set integer = 2
    num_set_values  10 10
    set_values 1 2 4 8 16 32 64 128 256 512   10:5:55
    descriptors  'c' 'd'

interface
  fork
    analysis_driver='case1_script.sh'
    parameters_file='params.in'
    results_file='results.out'

    #Tell Dakota to use every Dakota process to run simulations
    #(By default one is reserved for use as a master process)
    evaluation_scheduling peer dynamic

    #instructs Dakota to number files according to evaluation
    #number
    file_tag

responses
  descriptors 'x' 'y' 'z'
  objective_functions = 3
  no_gradients
  no_hessians
```

Figure 3: The Dakota input file ('case1.in') for the Use Case 1 example. This example runs a multidimensional parameter study for four parameters, each with 10 values investigated. Two of the parameters are real numbers, and two are integers. There are three response functions.

```
#!/bin/bash

#This script is given two arguments:
# - The name of the Dakota parameters file
# - The name of the Dakota results file

#get dakota job number
num=$(echo $1 | awk -F. '{print $NF}')
topdir=`pwd`
workdir=$topdir/workdir.$num

#Create working directory for simulation run
mkdir workdir.$num
cd $workdir

#pre-processing
dprepro $topdir/$1 $topdir/paramfile.template paramfile

#run simulation
/path/to/executable > sim_out.txt

#post-processing
#assume in this case the simulation's stdout is in the correct
#format for the Dakota file
cp sim_out.txt $topdir/$2

#delete working directory (optional)
cd $topdir
rm -rf $workdir
```

Figure 4: The interface script ('case1_script.sh') for the Use Case 1 example. The script produces a new work directory for each simulation, runs the simulation and returns its results to the Dakota results file before deleting the work directory.

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

#PBS -N Dakota_Case1
#PBS -l select=100
#PBS -l walltime=4:00:00
#PBS -A [budget code]

# Make sure any symbolic links are resolved to absolute path
export PBS_O_WORKDIR=$(readlink -f $PBS_O_WORKDIR)

# Change to the directory that the job was submitted from
# (remember this should be on the /work filesystem)
cd $PBS_O_WORKDIR

module load dakota

# Set the number of threads to 6
export OMP_NUM_THREADS=6

# Launch Dakota parallel jobs
#   100 nodes with 4 per node = 400 processes
#   6 cores per dakota process, 2 processes per NUMA Region
aprun -n 400 -N 4 -d 6 -S 2 dakota -i case1.in -o case1.out -e case1.err
```

Figure 5: The PBS submission script for the Use Case 1 example. The script selects 100 nodes, each with 4 Dakota processes running on them, resulting in a total of 400 Dakota processes (and hence 400 concurrent simulations).

3.2. Use Case 2: Optimisation Using a Genetic Algorithm

Assume we are designing a component whose design can be described by a number of parameters that must be optimised according to some constraint (for example, we want to choose the angle of attack and camber of an aircraft wing so as to maximise its lift to drag ratio). To calculate the properties of each design, a simulation that runs for around an hour on two nodes is used. With Dakota, we can use a genetic algorithm to find an optimal design for this component. Briefly, a genetic algorithm works by generating a number of designs, n, and evaluating their fitness via some metric. The algorithm picks a number, m, where ($m < n$), of the best designs from the original n, then combines the features of the m designs together and mutates them to produce a new generation of n designs. This process is then repeated for a number of generations until an optimal design has been found. For such an algorithm, Use Case 2 is suitable because many multi-node/many-processor simulations can be run concurrently (i.e. more than one member of each generation can be evaluated simultaneously).

Say (for the aircraft wing) we have two input parameters (angle of attack, a, and the camber, c) whose values can range between $[-10°, 25°]$ and $[0.0, 0.3]$ respectively. Although we wish to maximise the lift to drag ratio, r, when optimising, Dakota aims to minimise a response function, so in this case we want the response function to be $-r$. Let us choose a generation size of 50, and have the study run for a total of 6 generations. We choose to use 50 nodes for our study, resulting in 25 concurrent simulations.

There are three files we need to write: a Dakota input file, a script to interface the code with Dakota, and a PBS script to launch the job. The input file has to specify that we want Dakota to run several simulations concurrently, the interface script has to launch jobs using `aprun`, and the batch script has to run Dakota on a MOM node. Figure 6 shows the Dakota input file, Figure 7 shows the interface script, and Figure 8 shows the PBS script.

```
environment
  tabular_data

method
  max_iterations = 6
  max_function_evaluations = 500
  coliny_ea
    seed = 11011011
    population_size = 50
    fitness_type merit_function
    mutation_type offset_normal
    mutation_rate 1.
    crossover_type two_point
    crossover_rate 0.0
    replacement_type chc = 10

variables
  continuous_design = 2
    lower_bounds     -10    0.0
    upper_bounds      25    0.30
    descriptors      'a'     'c'

interface
    fork
    asynchronous
    evaluation_concurrency=25
    analysis_driver='case2_script.sh'
    parameters_file='params.in'
    results_file='results.out'
    file_tag

responses
  objective_functions = 1
  no_gradients
  no_hessians
```

Figure 6: The Dakota input file 'case2.in' for the Use Case 2 example. This example uses a genetic algorithm to optimise a wing design such as to maximise the lift to drag ratio. The algorithm investigates 6 generations of 50 mambers each. It is important to note that in this file we must specify 'asynchronous' and 'evaluation_concurrency = 25' in the 'interface' block in order to get Dakota to run multiple concurrent simulations.

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

#This script is given two arguments:
# - The name of the Dakota parameters file
# - The name of the Dakota results file

# Load your required programming environment (if necessary)
# module swap PrgEnv-cray PrgEnv-gnu

#get dakota job number
num=$(echo $1 | awk -F. '{print $NF}')

#set up topdir and workdir
topdir=‘pwd‘
workdir=$topdir/workdir.$num
mkdir workdir.$num

#move into workdir
cd $workdir

#pre-processing
dprepro $topdir/$1 $topdir/paramfile.template paramfile

#run simulation
aprun -n 48 -b /path/to/executable > sim_out.txt

#post-processing
#assume in this case the simulation produces a file 'out.dat'
#in the correct format for the Dakota file
cp out.dat $topdir/$2

#delete working directory (optional)
cd $topdir
rm -rf $workdir
```

Figure 7: The interface script (‘case2_script.sh’) for the Use Case 2 example. The script produces a new work directory for each simulation, runs the simulation (using aprun) and returns its results to the Dakota results file before deleting the work directory.

```
#PBS -N Dakota_Case2
#PBS -l select=50
#PBS -l walltime=6:00:00
#PBS -A [budget code]]

cd $PBS_O_WORKDIR

module load dakota

#run dakota on the MOM node (no aprun required)
dakota -i case2.in -o dakota.out -e dakota.error
```

Figure 8: The PBS script for the Use Case 2 example. Note that Dakota is run on the MOM node directly, and is not run in an aprun.

3.3. Use Case 3: Uncertainty Quantification

Assume we have a climate model where there is an uncertainty associated with two of the model's parameters (say the atmospheric concentrations of some aerosol particles), and we want to quantify how our uncertainty in the concentrations affects the results of our simulation. We can use Dakota's uncertainty quantification capabilities to achieve this. Using the Latin Hypercube Sampling method, Dakota evaluates the simulation results for a number of possible parameter values, and from that estimates the range of possible solutions given the uncertainty in the input parameters. Let us assume each simulation uses 50 nodes and takes around two hours to run. In this case, Use Case 3 is most suitable, whereby Dakota runs several consecutive 50 node simulations.

Let the two uncertain parameters be called p and q, where $p = 0.25 \pm 0.01$ and $q = 7.5 \pm 0.5$ (assuming the uncertainties are the standard deviation (σ) of a normal distribution). We will have Dakota sample 10 pairs of parameter values. We are interested in two response functions, t and u. Figure 9 shows the Dakota input file, Figure 10 shows the interface script, and Figure 11 shows the PBS script that we would use for this simulation.

```
environment
  tabular_data

method
  id_method = 'UQ'
  sampling
    sample_type lhs
    samples = 10
    seed = 98765 rng rnum2

variables
  normal_uncertain = 2
    means =           0.25   7.5
    std_deviations =  0.01   0.5
    descriptors  =    'p'    'q'

interface
    fork
    analysis_driver='case3_script.sh'
    parameters_file='params.in'
    results_file='results.out'
    file_tag

responses
  descriptors 't'  'u'
  response_functions = 2
  no_gradients
  no_hessians
```

Figure 9: The Dakota input file 'case3.in' for the Use Case 3 example. This example uses Latin Hypercube Sampling to determine the distribution of the simulation results t and u given the uncertainty in the input parameters p and q.

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

#This script is given two arguments:
# - The name of the Dakota parameters file
# - The name of the Dakota results file

# Load your required programming environment (if necessary)
# module swap PrgEnv-cray PrgEnv-gnu

#get dakota job number
num=$(echo $1 | awk -F. '{print $NF}')

#set up topdir and workdir
topdir=`pwd`
workdir=$topdir/workdir.$num
mkdir workdir.$num

#move into workdir
cd $workdir

#pre-processing
dprepro $topdir/$1 $topdir/paramfile.template paramfile

#run simulation
aprun -n 1200 -b /path/to/executable > sim_out.txt

#post-processing
#assume in this case we have written a small program (pps) that
#reads the simulation output files and writes the Dakota
#results to stdout
./pps > $topdir/$2

#delete working directory (optional)
cd $topdir
rm -rf $workdir
```

Figure 10: The interface script ('case3_script.sh') for the Use Case 3 example. The script produces a new work directory for each simulation, runs the simulation (using aprun) and returns its results to the Dakota results file before deleting the work directory.

```
#PBS -N Dakota_Case3
#PBS -l select=50
#PBS -l walltime=24:00:00
#PBS -A [budget code]]

cd $PBS_O_WORKDIR

module load dakota

#run dakota on the MOM node (no aprun required)
dakota -i case3.in -o dakota.out -e dakota.error
```

Figure 11: The PBS script for the Use Case 3 example. Note that Dakota is run on the MOM node directly, and is not run in an aprun.

3.4. Use Case 4: List Parameter Study

Finally, let us assume we have a very large simulation (say each run requires over half of ARCHER's nodes, and takes ~24h to run) for which we wish to investigate the results for a list of different parameters. In this case we would apply Use Case 4, whereby each simulation to be run is submitted by Dakota to the queue using qsub.

Let us assume we have 8 parameters, x_1 to x_8 and we wish to investigate 3 different points within the parameter space: $(x_1, ..., x_8) = (0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1)$, $(0.2, 0.5, 0.7, 0.8, 1.6, 0.2, 0.3, -0.7)$ and $(0.9, 1.0, 6.6, -10.0, 0.0, 3.0, 5.0, 0.0)$. We are interested in 7 response functions, y_1 to y_7.

In this case, we require a Dakota input file and two scripts (a pre-processing and submission script, and a post-processing script). Dakota is run once from the login node, with the 'analysis_driver' in the Dakota input file set to be the pre-processing script. This will cause Dakota to submit the simulations to the queue (and the scripts will return dummy response functions to Dakota). Once the simulations have finished running, we run Dakota again, but this time with the 'analysis_driver' set to be the post-processing script. This will make Dakota collect the results of the simulations and present them in a table. Figure 12 shows the Dakota input file, Figure 13 shows the interface script, and

Figure 14 shows the post-processing script.

```
environment
  tabular_data

method
  list_parameter_study
  list_of_points = 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1
                   0.2  0.5  0.7  0.8  1.6  0.2  0.3 -0.7
                   0.9  1.0  6.6 -10.0 0.0  3.0  5.0  0.0

variables
  continuous_design = 8
  descriptors = 'x1' 'x2' 'x3' 'x4' 'x5' 'x6' 'x7' 'x8'

interface
    fork
        #pre-processing (comment after sims have completed)
        analysis_driver='case4_script_pre.sh'
        #post-processing (uncomment after sims have completed)
        #analysis_driver='case4_script_post.sh'
        parameters_file='params.in'
        results_file='results.out'
        file_tag

responses
  descriptors 'y1' 'y2' 'y3' 'y4' 'y5' 'y6' 'y7'
  objective_functions = 7
  no_gradients
  no_hessians
```

Figure 12: The Dakota input file for the Use Case 4 example. When run initially, the analysis driver is to be set to 'case4_script_pre.sh' to submit the simulations to the queue. After the simulations have finished, set the analysis driver to 'case4_script_post.sh' to complete the analysis.

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

#This script is given two arguments:
# - The name of the Dakota parameters file
# - The name of the Dakota results file

#get dakota job number
num=$(echo $1 | awk -F. '{print $NF}')

#set up topdir and workdir
topdir=`pwd`
workdir=$topdir/workdir.$num
mkdir workdir.$num

#move into workdir
cd $workdir

#pre-processing
dprepro $topdir/$1 $topdir/paramfile.template paramfile

echo '#!/bin/bash --login' > submit.pbs
echo '#PBS -N dakota' >> submit.pbs
echo '#PBS -l select=3000' >> submit.pbs
echo '#PBS -l walltime=24:00:00' >> submit.pbs
echo '#PBS -A [budget code]' >> submit.pbs
echo 'cd $PBS_O_WORKDIR' >> submit.pbs
echo 'aprun -n 72000 /path/to/executable' >> submit.pbs

#submit job
qsub submit.pbs

#return dummy values to Dakota
echo '1.0 1.0 1.0 1.0 1.0 1.0 1.0' > $topdir/$2
```

Figure 13: The job submission interface script for the Use Case 4 example, 'case4_script_-pre.sh'. This script produces a PBS submission script, submits the job and then returns dummy values to Dakota via the Dakota results file.

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

#This script is given two arguments:
# - The name of the Dakota parameters file
# - The name of the Dakota results file

#get dakota job number
num=$(echo $1 | awk -F. '{print $NF}')

#set up topdir and workdir
topdir=‘pwd‘
workdir=$topdir/workdir.$num

#move into workdir
cd $workdir

# Post-processing of simulation results
# (Assume simulation produces an output file, 'out.dat'
# compatible with the Dakota results file)
cp out.dat $topdir/$2

# Delete working directory (optional)
cd $topdir
rm -rf $workdir
```

Figure 14: The simulation results post-processing script for the Use Case 4 example, ‘case4_script_post.sh’. This script takes the output from the simulation and returns it to Dakota via the Dakota results script.

4. Tips and Recommendations

We will now outline some general tips to help you run Dakota more effectively on ARCHER. In particular we will offer suggestions and guidelines for running Dakota on the MOM nodes so as to prevent any potential disruption to other users of the ARCHER service.

4.1. General Tips

- Writing a Dakota input file can be tricky due to the Dakota User's[4] and Reference[3] Manuals being confusing. In order to prevent wasting CPU time and time waiting in the queue, only for your Dakota study to fail due to an incorrect input file, it is recommended to try out your Dakota input file (using a dummy simulation interface script) on your local machine. Dakota binaries can be found on the Dakota website for Windows, OS X/macOS and Linux, and are suitable for running on a laptop.

- In order to simplify the post-processing process, it is advisable - where possible - to modify your simulation code to output a file containing the response functions needed by Dakota. This simplifies the interface script considerably, and also reduces the CPU time cost on the MOM nodes (see below).

- To carry out a speedup analysis or a similar study where the number of processes each simulation is to be run on is a parameter to be varied by Dakota, use Use Case 4 and produce your job submission scripts with `dprepro`.

- Make sure that your interface script is executable - i.e. `chmod +x script.sh`.

4.2. Running Dakota on the MOM Nodes

The MOM nodes are a shared resource on ARCHER, and as such using too much CPU time on them can make your job subject to automatic cancellation. Here are some tips for how to minimise CPU usage on the MOM nodes:

- In the `aprun` command in your interface script, use the `-b` argument, which prevents the MOM node from compressing your executable and transferring it to the compute nodes. Instead, the compute nodes will use the path of the executable as specified in the `aprun` command. *Note: your simulation executable must be located on the work filesystem as the compute nodes cannot access the home filesystem.*

- There is a limit of 500 simultaneous `apruns` per job on ARCHER. Please ensure, therefore, that no more than 499 concurrent simulations are taking place in order to avoid exceeding this limit.

- Please try to keep the pre/post-processing CPU time in the interface script to a minimum. This could be achieved by incorporating a lot of the work into your simulation code (which runs on the compute nodes), or by running any post/pre-processing on compute nodes via `aprun`.

- Try to avoid simulations that take a very short time to run (seconds to a few minutes) as a lot of CPU time will be used on the MOM nodes to launch the simulations and copy files. For such simulations, Use Case 1 is more appropriate.

References

[1] *Dakota Website*, (Accessed 22 Mar 2017). URL https://dakota.sandia.gov.

[2] *Dakota Developer's Manual*, (Accessed 22 Mar 2017. URL https://dakota.sandia.gov/content/64-developers-manual.

[3] *Dakota Reference Manual*, (Accessed 22 Mar 2017). URL https://dakota.sandia.gov/content/64-reference-manual.

[4] Brian M. Adams, Mohamed S. Ebeida, Michael S. Eldred, John D. Jakeman, Kathryn A. Maupin, Jason A. Monschke, Laura P. Swiler, J. Adam Stephens, Dena M. Vigil, and Timothy M. Wildey. *Dakota User's Manual*, May 9 2016. URL https://dakota.sandia.gov/sites/default/files/docs/6.4/Users-6.4.0.pdf.