Optimizing Alya simulations of multi-physics problems with Dakota

Rogeli Grima

December 15, 2014

Contents

1 Introduction

This user's guide is intended to provide some background information on how to use Dakota to solve optimization problems that involve a simulation with Alya. We assume the user has some familiarity with Alya execution and configuration.

In this guide we will show how to use Dakota in Marenostrum, how to prepare your simulations, how to process their outputs and we will provide some basic examples.

1.1 What is Dakota?

Dakota is a toolkit that provides a flexible, extensible interface between analysis codes and iterative systems analysis methods. It contains algorithms for optimization, uncertainty quantification, parameter estimation and sensitivity/variance analysis.

Many computational methods are used to simulate complex physical systems. On many occasions we want to use these simulations to find an acceptable or optimized solution for a particular system. Dakota will help us to use a computational method as a design tool. It will help us to solve very important questions as: What is the best design? How safe is it? How much confidence do I have in my answer?

The Dakota Software Toolkit is being developed in the Sandia National Laboratories. It is written in C++ and it can be used with programs written in any language.

1.2 Dakota capabilities

Dakota delivers a variety of iterative methods and strategies, and the ability to flexibly interface them to a simulation code. While Dakota was originally conceived to more readily interface simulation codes and optimization algorithms, recent versions include other iterative analysis methods such as uncertainty quantification with nondeterministic propagation methods, parameter estimation with nonlinear least squares solution methods, and sensitivity/variance analysis with general-purpose design of experiments and parameter study capabilities. These capabilities may be used on their own or as building blocks within more sophisticated strategies such as hybrid optimization, surrogate-based optimization, optimization under uncertainty, or mixed aleatory/epistemic UQ.

These are the principal classes of Dakota algorithms:

- Parameter studies.
- Design of experiments.
- Uncertainty Quantification.
- Optimization.
- Calibration.

2 Dakota details

2.1 Dakota workflow

Dakota receives input and configuration from a variety of text files prepared by the user, and connects to any simulation code by other text files. This simple interface is one of the most important features of Dakota, as it makes it easy to change the iterative method or strategy by just changing a few lines in the Dakota input file.

In Figure 1, we can see the Dakota workflow. The user should provide an input file to Dakota. This input file controls the algorithm that we want to run, the function to be evaluated, the variables of this function and the outputs that we expect.

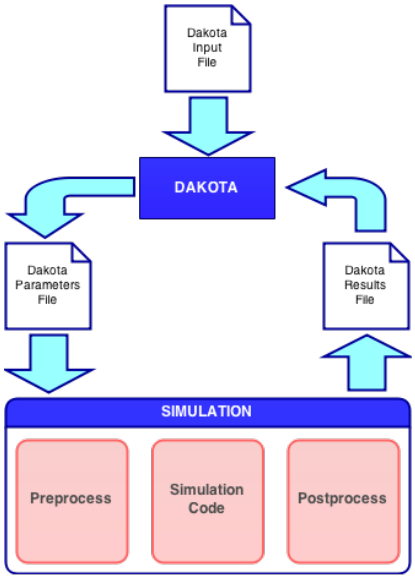

Figure 1: Dakota Workflow

Dakota generates a parameter file for every function evaluation. This file contains a specific value of every variable and specifies the kind of returning values that it needs (function evaluation, derivatives or/and second derivatives).

Dakota treats the simulation code as a black box. Most of the commercial codes don't know what to do with the parameter file, so it is common to create a script to connect the simulation code with Dakota. In this script we merge the Dakota parameters file with the simulation code input files.

Once we have some proper input files we can run the simulation code, extract the significant data from its output files and post-process that data in order to generate the appropriate objective function and the Dakota results file.

2.2 Dakota files

In order to run Dakota we must provide an input file. This is the typical way to run it:

```
dakota -i input_file -o output_file
```

Meanwhile, Dakota provides two files to the simulation code. One is for setting the variables and the other is to receive the outputs. This is an example of how Dakota runs a simulation code:

```
./reactor_alya.sh params.in results.out
```

Dakota input file format

There are six sections in every Dakota input file. These sections are identified with the following keywords: variables, interface, responses, model, method, and environment. At least one variable, interface, responses, and method must appear, and no more than one environment should appear.

3 Using Dakota in Marenostrum

We provide the last three versions of Dakota. You can find them in:

- /gpfs/projects/bsc21/DAKOTA/dakota-5.4.0
- /gpfs/projects/bsc21/DAKOTA/dakota-6.0.0
- /gpfs/projects/bsc21/DAKOTA/dakota-6.1.0

Original code from Dakota has an incompatibility with the queue system of Marenostrum. The queue system kills all the asynchronous task that Dakota uses to run concurrent jobs. We have modified Dakota code to avoid this problem. For more information see Appendix A

Althought, most of the experiments that we have ran have been done using version 5.4 we encourage you to use the last version. From now on, all

the information that we will provide you in this guide will refer to version 6.1.

In order to run Dakota you will need to set the path of the executables and the directory of the shared libraries. You can add these lines to your .barshrc file:

```
export DAK_INSTALL=/gpfs/projects/bsc21/DAKOTA/dakota-6.1.0
export PATH=${DAK_INSTALL}/bin/:${PATH}
export LD_LIBRARY_PATH=${DAK_INSTALL}/bin:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${DAK_INSTALL}/lib:${LD_LIBRARY_PATH}
```

Check that everything is working properly by running:

```
dakota -v
```

A Dakota Source code modification

In the Linux Bash terminal, an orphan process can be created by attaching an ampersand at the end of the command line. This is an abstract of the original file CommandShell.cpp where we can see how Dakota runs asynchronous jobs:

```
CommandShell& CommandShell::flush()
{
  if (asynchFlag)
    sysCommand += " &";
  std::system(sysCommand.c_str());
}
```

As we can see, Dakota makes a system call using the ampersand when the asynchronous flag is true. In this case, the new process is orphan and this can became a problem. If we want to avoid this, we can not use the ampersand.

```
CommandShell& CommandShell::flush()
{
  if (asynchFlag)
  {
    if ( fork() == 0 )
    {
      std::system(sysCommand.c_str());
      exit( 0 );
    }
  }
  else
    std::system(sysCommand.c_str());
}
```

In the new version, we use threads, forking a new DAKOTA process for each new simulation. We can identify the original process because the returning value of fork is different from zero. This process exits the function without doing anything. Meanwhile, the new process (the returning value of fork is equal to zero) makes the system call to execute the task, but without detaching it. The task is never an orphan process. Once the task is finished the process is terminated.