Optimizing Alya simulations of multi-physics problems with Dakota

Rogeli Grima

December 15, 2014

Contents

1 Introduction

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.

Alya is a computational method that is 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 Alya 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?

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 with Alya, how to process their outputs and we will provide some basic examples.

1.1 Dakota workflow

Before you start working with Dakota and Alya, is important to know the Dakota's workflow. This will give you a better perspective of what is happening.

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.

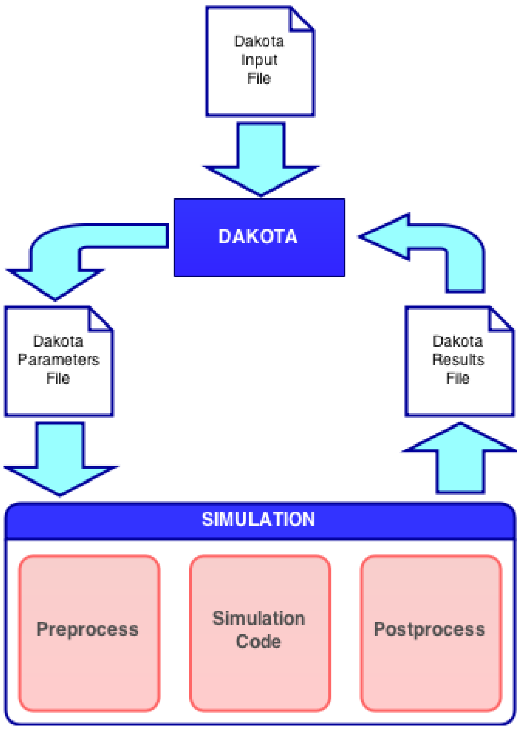

Figure 1: Dakota Workflow

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 codes, like Alya, 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 Dakota tutorial

2.1 Set up Dakota

We have installed in Marenostrum the last three versions of Dakota. 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. You can find the several installed versions of Dakota 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

If you want to install your own version of Dakota in Marenostrum, please check Appendix A.

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
```

2.2 Runing Dakota with a simple input file

This section is intended for users who are new to Dakota, to demonstrate the basics of running a simple example.

1. Create a working directory.
2. From path /gpfs/projects/bsc21/DAKOTA/Examples/power, copy files power_multidim.in and power.py to the working directory.
3. From the working directory, run: dakota -i power_multidim.in -o power_multidim.out > power_multidim.stdout

Dakota outputs a large amount of information to help users track progress. Four files should have been created:

1. The screen output has been redirected to the file power_multidim.stdout. The contents are messages from Dakota and notes about the progress of the iterator (i.e. method/algorithm).
2. The output file power_multidim.out contains information about the function evaluations.
3. power_multidim.dat is created due to an specification of the input file. This summarizes the variables and responses for each function evaluation.
4. dakota.rst is a restart file. If a Dakota analysis is interrupted, it can be often be restarted without losing all progress.

This example used a parameter study method and the power.py test problem. The Python script power.py reads a Dakota parameters file, computes the function $F(x, y) = (X - 0.5)^2 + (Y + 0.5)^2$ and writes the result in a Dakota results file. You can see a plot of the results executing the script:
/gpfs/projects/bsc21/DAKOTA/Examples/power/visualize.sh.

A Dakota Source code modification

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.

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 call fork to create 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 (its returning value from 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.