The DeBasher Software Package

Daniel Ortiz Martínez

Mathematics and Computer Science Department

University of Barcelona

Table of Contents

1. Introduction

2. Package Overview

3. Main Tools and File Formats

4. Toy Program Example

Introduction

Flow-Based Programming

- Flow-based programming models a program as a network of components which communicate by sending and receiving data through predefined connections
- `DeBasher` is a flow-based programming extension for Bash

Pipeline Execution

- Pipeline execution is a complex task
    - Pipeline composed of very heterogeneous tasks/processes
    - Processes may present dependencies with other ones
    - Often necessary to add or remove pipeline processes
    - Need to allocate computational resources
    - Independent processes should be executed concurrently
    - Hard to maintain and reuse code
    - ...
- `DeBasher` can be useful for modular and scalable pipeline execution

Package Overview

Package Dependencies

- Shell Bash
- Python
- Slurm Workload Manager (optional)

Package Installation

- Obtain the package using git:

```
git clone https://github.com/daormar/debasher.git
```

- Change to the directory with the package's source code and type:

```
./reconf
./configure
make
make install
```

NOTE: use `--prefix` option of `configure` to install the package in a custom directory

Functionality

- DeBasher is an engine to execute general programs
- DeBasher is particularly useful to execute pipelines
- Executes only those program processes that are pending
- Handles computational resources for each process
- Executes process arrays

Execution Model

- DeBasher follows the *flow-based programming* paradigm
  - Network of *black box* processes
  - Relations between processes are defined by the data they exchange
  - Component oriented
- DeBasher follows a simple execution model based on a file enumerating a list of program processes to be executed
- Processes are executed simultaneously unless dependencies are specified
- Process implementation is given in module files

Main Tools and File Formats

Main Tools

- `debasher_exec`
- `debasher_exec_batch`
- `debasher_status`

debasher_exec

- Automates execution of general programs
- Main input parameters:
  - `--pfile <string>`: file with processes to be executed
  - `--outdir <string>`: output directory
  - `--sched <string>`: scheduler used for program execution
  - `--show-cmdline-opts`: show command line options
  - `--check-proc-opts`: check process options
  - `--debug`: do everything except launching processes

`debasher_exec`: Output

- Content of output directory:
  - `__scripts__`: directory containing the scripts used for each process
  - `<process_name>`: directory containing the results of the process of the same name
- Additional directories may be created depending on the program

`debasher_exec`: Available Schedulers

- **Built-in Scheduler**
    - Allows to execute programs locally
    - Incorporates a basic resource allocation mechanism
- **Slurm Scheduler**
    - Allows to exploit large computational resources
    - Usage transparent to the user
    - Slurm behavior influenced by program description

debasher_exec_batch

- Automates execution of program batches
- Main input parameters:
  - `-f <string>`: file with a set of `debasher_exec` commands
  - `-m <string>`: Maximum number of concurrently executed programs
  - `-o <string>`: Output directory to move output of each program

debasher_status

- Checks execution status of a given program
- Main input parameters:
  - `-d <string>`: directory where the program processes are stored
  - `-s <string>`: process name whose status should be determined (optional)

The `debasher_lib.sh` Library

- Shell library with functions used by the previously described tools
- Functions can be classified as follows:
  - Implementation of the package execution model
  - Automated creation of scripts executing processes
  - Helper functions to implement processes

File Formats

- **Module file**: file defining the code of the program processes. Module files can also define a program by enumerating the processes involved
- **Program automation script**: file with a sequence of `debasher_exec` commands automating the analysis of a dataset

Module File

- Contains the definition of the different processes
- Written in `bash`
- Three `bash` functions should be defined for each process:
    - `processname_explain_cmdline_opts()`
    - `processname_define_opts()`
    - `processname()`

Module File: `processname_explain_cmdline_opts()`

- This function documents the command line options that the process needs to work
- The aggregated documentation for the different processes is shown when executing `debasher_exec --showopts`
- Whenever two processes share the same option, it is important to give it the same name

Module File: processname_explain_cmdline_opts()

```
process_a_explain_cmdline_opts()
{
    # -a option
    description="Sleep time in seconds for process_a (required)"
    explain_cmdline_opt "-a" "<int>" "$description"
}
```

Module File: `processname_define_opts()`

- This function should create a string containing the options that are specific to the process
- The main idea is to map command line options to process options
- The package provides multiple built-in functions to make the implementation of this function easier

Module File: processname_define_opts()

```
processname_define_opts()
{
    # Initialize variables
    local cmdline=$1
    local process_spec=$2
    local optlist=""

    # Use built-in functions to add options to optlist variable
    ...

    # Save option list
    save_opt_list optlist
}
```

Module File: processname_define_opts()

```
process_a_define_opts()
{
    # Initialize variables
    local cmdline=$1
    local process_spec=$2
    local process_name=$3
    local process_outdir=$4
    local optlist=""

    # -a option
    define_cmdline_opt "$cmdline" "-a" optlist || exit 1

    # Save option list
    save_opt_list optlist
}
```

Module File: `processname()`

- Implements the process
- The function should incorporate code at the beginning to read the options defined by `processname_define_opts()`

Module File: processname()

```
process_a()
{
    # Initialize variables
    local sleep_time=`read_opt_value_from_line "$*" "-a"`

    # Sleep some time
    sleep ${sleep_time}
}
```

Module File: Program Definition

TO-BE-DONE

Program Automation Script

- Automates the analysis of a whole dataset
- At each entry (one per line), `debasher_exec` tool is used to execute a whole program
- Can be used as input for `debasher_exec_batch`
- Entry example:

```
debasher_exec --pfile example.ppl --outdir outdir1 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...
debasher_exec --pfile example.ppl --outdir outdir2 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...
debasher_exec --pfile example.ppl --outdir outdir3 --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...
...
debasher_exec --pfile example.ppl --outdir outdirn --sched SLURM -opt1 <opt1_val> -opt2 <opt2_val> ...
```

Extending Modules

- Since multiple modules can be loaded, a new module may contain process definitions missing in another one
- The order in which modules are imported is relevant
  - if two modules define the same function, the definition in the module imported last will prevail
  - the previous property can be used to modify a specific process without repeating the code of the whole module

Toy Program Example

Pipeline File

TO-BE-DONE

Program Representation

TO-BE-DONE