IBM Appresentor: automatize the generation, compilation, and profiling of an experimental campaign

January 29, 2018

Contents

1 Introduction

IBM Appresentor is a design-of-experiments toolchain that automates most of the work required to generate a set of experiments to be profiled with the *IBM Platform-Independent Software Analysis* tool [1]. It includes makefiles implementing the compilation and instrumentation flow, assuming that *IBM Platform-Independent Software Analysis* has been installed and its environment variables are set correctly. Additionally, it includes scripts to generate a set of experiments for a target application. These experiments are obtained by varying the set of application parameters. Once the experiment set is generated, compilation instrumentation and execution of all experiments can be booted using commands defined in the makefiles.

The toolchain supports C, C++ and, when using LLVM 3.5.0 and dragonegg, FORTRAN. Parallel applications implemented in OpenMP and MPI are supported. Parallel applications implemented using other paradigms are not supported.

2 Requirements

The scripts to generate the design of experiments require Python. The toolchain has been tested with Python 2.7.14 and Python 2.7.5. The pyDOE package is also required. *IBM Platform-Independent Software Analysis* should be installed along with all its requirements, including LLVM. For FORTRAN applications LLVM 3.5.0, dragonegg, and gcc are required. The toolchain was tested with gcc 4.8.5.

All the requirements of the target application should also be met.

3 Usage

To use the toolchain you can proceed with the following steps:

1. If the target application is not released with the DOE-toolchain distribution, install the target application in the benchmark folder (Section 3.1).
2. Generate the experiment set (Section 3.2).
3. Profile the experiments by compiling and running them (Section 3.2).

3.1 Installing a new benchmark

Let `<DOE-root>` be the folder where the DOE-toolchain can be found. To install a new application, first of all create a folder:

`<DOE-root>/benchmarks/<suiteName>`,

where `<suiteName>` names your benchmark suite. When generating experiments (Section 3.2), you will have to choose a benchmark suite, all applications within that suite will be processed. Then, create the application folder using the application name `<appName>`:

`<DOE-root>/benchmarks/<suiteName>/<appName>`.

Within this folder, create the following two subfolders:

- `autorun_config`
- `src`

In `autorun_config`, the definition of the application parameters to be handled by the DOE toolchain must be defined as described in Section 3.1.1. The whole source code of your application should be deployed in the folder `src`. Source files should be named using extensions that are representative of the source language. Use `.c` for C files, `.cpp` or `.cc` for C++ files, and `.f` for FORTRAN files.

The *IBM Platform-Independent Software Analysis* instrumentation is carried out after the linking phase. First, all source files will be compiled to generate LLVM IR files named by replacing the file source file extension with `.bc`. Second, all these files are linked together into an IR file named `main.bc`. This file `main.bc` will be instrumented using *IBM Platform-Independent Software Analysis*. No source file should be named `main.c` (or any other extension) because its IR file name would conflict with the overall linked program `main.bc`.

We recommend to read the parameter passing mechanism described in Section 3.1.2 to clarify how the DOE toolchain will pass compile-time and run-time parameters.

3.1.1 Parameter space definition

The DOE toolchain is meant to automatize the generation and execution of several experiments for a target application. Experiments are executed by varying some compile-time and run-time parameters. Parameters and parameter spaces must be defined in the file:

`<DOE-root>/benchmarks/<suiteName>/<appName>/autorun_config/autorun.py`,

Different parameter spaces can exist for a single application. For example one may want to have an experimental space for short runs and one for longer runs. Each parameter space has its own name `<paramSpace>` and must be defined in `autorun.py`. When generating the experiments (Section 3.2) you should specify the name `<paramSpace>` of the space you want to use. We explain the format of `autorun.py` with an example:

```
########## example configuration file ##########
def getInputParameterRange(dsName):
    """
    This Python function defines the input parameter spaces for the
    given application.
    Parameters:
        dsName: the name of the parameter space.
    Return:
        build_input
            A dictionary listing the compile-time parameters:
                - key -> string, the name of the parameter
                - values -> a list of strings representing the parameter values
        run_input
            A dictionary listing the run-time parameters:
                - key -> string, the name of the parameter
                - values -> a list of strings representing the parameter values
        openmp_threads
            The list of parallelism options for OpenMP applications
        mpi_procs
            The list of parallelism options for MPI applications
    """

    found = False;

    if(dsName == "test"):
        found = True
        print('[INFO] DOE \"test\" found!')
        build_input = {
        'PROBLEMSIZE'   : [ '33' , '36' , '39' , '42' , '45' ],
        'IDT'       : [ '0.25' , '0.75' , '1.25' , '1.75' , '2.25' ],
        'NITER'     : [ '9' , '12' , '15' , '18' , '21' ],
        'PROCS'     : [ 'mpi_procs' ]
        }

        run_input = {
        }

        omp_threads = [] # an empty dictionary means no OpenMP

        mpi_procs = ['25' , '36', '49', '64' , '81' ]

    if(dsName == "alternativeTest"):
        found = True
        print('[INFO] DOE \"networkTest\" found!')
        build_input = {
        'PROBLEMSIZE'   : [ '36' , '64' , '102' , '162' ],
        'IDT'       : [ '0.5' ],
        'NITER'     : [ '30' ],
        'PROCS'     : [ 'mpi_procs' ]
        }

        run_input = {
        }

        omp_threads = [] # an empty dictionary means no OpenMP

        mpi_procs = ['9' , '16' , '25' , '36' , '49' , '64' ]

    ###################### default, "train
    if(not found):# default, "train"
        print('[INFO] DOE not found, using DOE \"train\"!')
        build_input = {

        'PROBLEMSIZE'   : [ '15', '18', '21', '24', '27' ],
        'IDT'       : [ '0.25' , '0.5' , '0.75' , '1.0' , '1.25' ],
        'NITER'     : [ '3' , '6' , '9' , '12' , '15' ],
        'PROCS'     : [ 'mpi_procs' ]
        }

        run_input = {
        }

        omp_threads = []

        mpi_procs = ['9', '16', '25', '36', '49']

    return build_input, run_input, omp_threads, mpi_procs


# This function was meant for polyhedral optimizations and is experimental.
# It defines the dimension of the used data structures.
# Do not modify it.
def getDimensionNumber():
    return 0
```

The file `autorun.py` defines the Python function `getInputParameterRange` that takes as input the name of the parameter space and returns as output the requires space. In the example there are defined three spaces: `'test'` (definition begins at line 25), `'alternativeTest'` (definition begins at line 42), and `'train'` (definition begins at line 60). If the parameter space name is not found, the `'train'` space is returned.

This function returns as output four variables defining the parameter space: `built_input`, `run_input`, `omp_threads`, and `mpi_proces`. The first part describes the compile-time parameters, the second part describes the run-time parameters and the last two define the parallelism levels for OpenMP and MPI. When declaring parameters in `built_input`, `run_input` we recommend not to use the `_` character in parameter names. This character will generate problems if the profiles are then processed using *IBM Exascale Extrapolator* because `_` is a special character in the Mathematica language.

3.1.2 Parameter passing

The application parameters are passed from the DOE toolchain to the application in different ways depending on the type of parameter.

Compile-time parameters.

Compile time parameters defined in `build_input` (in the file `autorun.py`) are passed through macros handled by the C preprocessor. The macros are defined when clang (or gcc) is invoked by passing the flags `-D<paramName> <paramValue>` for each parameter listed in the dictionary `build_input`. The values `<paramValue>` are varied from experiment to experiment but are always taken from within the list associated to the key `<paramName>` in the dictionary `build_input`.

NOTE: The macro definitions `-D<paramName> <paramValue>` at compile time are handled by the C preprocessor. FORTRAN files will be compiled to LLVM IR by using dragonegg (a gcc plugin). The compile time parameters are still passed as C preprocessor macros as defined previously. Take care that FORTRAN code might include several `include` statements that are not managed by the C preprocessor. Bugs may arise. Note that, since gcc is used for compiling the FORTRAN files, the C preprocessor is executed also for FORTRAN code. You can replace any `include` statement with the C preprocessor statement `#include` if needed.

Run-time parameters.

The toolchain will also run the application for you considering all the settings required by the selected design of experiment technique. Each run-time parameter is passed to the application at runtime using two consecutive arguments: `<paramName> <paramValue>`. There will be an argument couple for each parameter `<paramName>` defined as a key of the dictionary `run_input` returned by the function `getInputParameterRange`. When automatically invoked the application executable will take only these arguments. **No other run-time argument will be passed to the application when automatically executed by the DOE toolchain.** The application code may need to be editid to fit in this parameter-passing mechanism.

Parallelism parameters

The variables `omp_threads`, and `mpi_proces` returned by `getInputParameterRange` define the parallelization options for OpenMP an MPI respectively. These variables must be a list of natural number, leave an empty list if the application does not use OpenMP or MPI. The actual parallelism to use may vary from experiment to experiment and it is automatically identified by the DOE toolchain among the ones in the returned list.

When an experiment is run, the number of OpenMP thread is passed as an environment variable while the number of MPI processes is passed when automatically invoking `mpirin`.

The toolchain enable to pass the value of parallelism parameters also as compile-time or run-time parameter. If you want a parameter `<paramName>` to always assume the value of the parallelism in use, define the list of possible values such to include a single value that refers to the parallelism parameter name. For example in the `autorun.py` file listed above, the compile time parameter `PROCS` will always assume the same value of the number of MPI processes in use.

3.1.3 Setting up the Makefile

All the source files that need to be compiled must be specified in an application Makefile:

`<DOE-root>/benchmarks/<suiteName>/<appName>/Makefile`.

This file must also include the list of the compilation, instrumentation and execution commands defined within the DOE toolchain, and the list of *IBM Platform-Independent Software Analysis* instrumentation flags. Following an example of this Makefile:

```
include ../../../../../Makefile.common


############ GENERAL CONFIGURATION


APP_NAME=LU
LANGUAGE=FORTRAN

SOURCES_C=
SOURCES_CC=
SOURCES_CPP=
SOURCES_F= bcast_inputs.f   error.f exchange_3.f \
                        exchange_6.f  jacu.f neighbors.f  proc_grid.f \
                        setbv.f ssor.f  buts_vec.f  exact.f \
                        exchange_4.f  init_comm.f  l2norm.f  nodedim.f \
                        read_input.f  setcoeff.f  subdomain.f \
                        blts_vec.f  erhs.f  exchange_1.f  exchange_5.f \
                        jacld.f  lu.f  pintgr.f  rhs.f  setiv.f  \
                        verify.f timers.f print_results.f

############ COMPILE-TIME APPLICATION CONFIGURATION
```

```
# Default values of compile-time parameters
PROBLEMSIZE?=32
DT?=1
NITER?=60
PROCS?=9

############ ADDITIONAL COMPILER FLAGS
APPADDFFLAGS=\
        -DPROBLEMSIZE=$(PROBLEMSIZE) -DDT=$(DT) \
        -DNITER=$(NITER) -DPROCS=$(PROCS) \
        -I$(MPI_DIR)/include
APPADDLDFLAGS=-L$(MPI_DIR)/lib -lmpi -lmpi_mpifh


include ../../../../../Makefile.common.PISA
include ../../../../../Makefile.common.commands
```

For a new application you may copy the above content and edit the following definitions:

- `APP_NAME`: the name to be given to the executable application executable.
- `LANGUAGE`: the source-code language (either `C`, `C++`, or `FORTRAN`).
- `SOURCES_C`: the list of C source files to be compiled.
- `SOURCES_CC`: the list of C++ source files terminating with the extension `.cc` to be compiled.
- `SOURCES_CPP`: the list of C++ source files terminating with the extension `.cpp` to be compiled.
- `SOURCES_F`: the list of FORTRAN source files to be compiled.
- The application compile-time parameters (in the example: `PROBLEMSIZE`, `DT`, `NITER`, and `PROCS`). Assign default values to these variables by using the operator `?=` such to enable the DOE toolchain to override their definition when the Makefile will be invoked.
- `APPADDFFLAGS`: the application specific flags to be passed at compile time. Together with the compile-time flags, you also need to specify the compile-time parameter passing mechanism (i.e. the macro definitions).
- `APPADDLDFLAGS`: the application-specific flags to be passed to `clang` when linking.

3.2 Setup and run an experimental campaign

All the scripts automating the setup of an experimental campaign are located in the folder: `<DOE-root>/scripts`. A design of experiment for a preinstalled benchmark suite can be generated from within the `scripts` directory by invoking the `experiment_generator` script. You can execute the command: `./experiment_generator --help` for a quick help. The script has to be invoked with the following parameters:

```
> ./experiment_generator --benchmark <suiteName> \
    --designType <doeName> --designSpace <paramSpace>
```

Where `<suiteName>` is the name of the benchmark suite target of the experimental campaign, `<doeName>` is the design of experiments (DOE) to apply (one amongst the DOE listed in Section 3.2.1), and `<paramSpace>` is the name of the design space defined in the applications configuration scripts `autorun.py`. The script will generate the folder `<DOE-root>/results/experiment/<timestamp>` that contains a Makefile setup to generate and profiles all experiments for all applications. It also contains a folder for each application in the benchmark suite. Instructions to compile all the experiments and run them are listed on the standard output:

```
[INFO] In order to run the experiment campaign:
[INFO]    1) cd <DOE-root>/results/experiment/<timestamp>
[INFO]    2) make generate.cnls -j20
[INFO]    3) make run.cnls -j8
```

The make command `generate.cnls` compiles and instruments the application for all the compile-time configurations defined by the DOE and the parameter space. The make command `run.cnls` executes all the experiments for the required parameter configurations using the instrumented binaries. You can find all the *IBM Platform-Independent Software Analysis* profiles with the command:

```
> find . -name '*.rawprofile'
```

If you are interested to profile a single application within the benchmark suite, use the make commands `gen<appName>.cnls` and `run<appName>.cnls`. You can also profile the native execution of each experiment by invoking

```
> make runnative
```

or

```
> make runnative<appName>
```

You will find the set of files storing execution time information with the command:

```
> find . -name '*.time'
```

3.2.1 Design of experiments

The toolchain supports the six following different design of experiments [2]:

- `Plackett-Burman`: it is a supersaturated DOE based on two parameter levels. For any parameter `<parName>` in the compile-time, run-time or parallelism parameter whose definition in `autorun.py` includes more than two levels, the first and last one are used.

- `Two-Level-Full-Factorial`: it consider that each parameter can assume either a high or a low value. For any parameter `<parName>` in the compile-time, run-time or parallelism parameter whose definition in `autorun.py` includes more than two levels, the first and last one are used.
- `Central-Composite`: it is a design of experiments for curve fitting and it considers that each parameter can assume one out of five values {*minimum, low, central, high, maximum*}. It requires that the parameter space definition in `autorun.py` lists exactly five values for each parameter (including compile-time, run-time and parallelism parameters).
- `Latin-Square`: it is a space filling technique that assumes each parameter defined in `autorun.py` to have the same number of possible values.
- `Exhaustive`: it consider experiments to cover all possible parameter value combinations, i.e. it covers the whole parameter space.

4 Contributors

- Davide Gadioli
- Giovanni Mariani

5 Acknowledgement

This work is conducted in the context of the joint ASTRON and IBM DOME project and is funded by the Dutch Ministry of Economische Zaken and the Province of Drenthe.

References

[1] A. Anghel, L. M. Vasilescu, G. Mariani, R. Jongerius, and G. Dittmann, "An instrumentation approach for hardware-agnostic software characterization," *International Journal of Parallel Programming*, pp. 1–25, March 2016.

[2] D. Montgomery, *Design and Analysis of Experiments, 8th Edition.* John Wiley & Sons, Incorporated, 2012.