Allscale AMDADOS Application Technical Report

Albert Akhriev
Add Yourself

April 10, 2018

Contents

1 Overview

2 Working with Amdados application

2.1 Building the application.

1. The application executable must be available before running any test (even in Python).

2. We recommend the standard way for building the application presented in the script `standard.build.sh` in the project root folder.

3. Another useful script `./scripts/download.sh` downloads the latest Allscale API and the Armadillo library for unit tests.

4. We do *not* recommend the development script for building the application `mybuild`, which relies on ramdisk and other development specific features. For completeness, we provide script's options: `-f` clears any previous build and starts from scratch; `-r/-d` release/debug mode; `-t` runs tests after building the project. For example: `./mybuild -f -r -t`.

2.2 Application parameters.

1. The application is controlled by configuration file. The default one can be found in the project root folder under the name `amdados.conf`. The most interesting parameters are: the integration period and the number of sub-domains in either dimension.

2. Not everything can be controlled by configuration file. For example, the size of sub-domain is hard-coded because of using of templates in C++ grid implementation.

3. The flow model is also hard-coded (see the function `Flow()` in the file `scenario_simulation.cpp` and corresponding function in the Python code). The reason for that is to get away of any specific format of flow data representation while focusing solely on Allscale API.

4. The same is true for sensor locations. Currently, user has to specify a fraction of nodal points occupied by sensors and then the location are (pseudo) randomly generated.

5. The default configuration file `amdados.conf` contains brief description of each parameter sufficient to match it to the source code.

2.3 The application structure

1. In order to demonstrate data assimilation process we need a "true" solution or, in other words, a "true state of nature". The reason is twofold. First, the *input data* "measured" at sensor locations simulate the process

of data acquisition in real world applications. The **observations** collected across the domain govern the simulation by pushing it towards the "true" solution. Second, the ground-truth solution is used for the accuracy assessment of data assimilation.

2. In the use-case scenario, named "simulation", *input data are simulated* by direct forward solver (see equation (**??**)) in the entire domain. The solver is written in Python. For simplicity, the Python forward solver does not implement any sub-domain partitioning, operating in the entire domain directly, but uses the same number of nodal points as C++ Amdados application, the same integration period and the same flow model, see the functions `Flow()` in both C++ and Python implementations.

3. The Python code, implemented in `python/ObservationsGenerator.py`, reads the configuration file (assuming the C++ application will use the same file later on) to set-up geometry and initial conditions. Then it runs the forward solver producing two large files.

4. Those files are: (1) values of the state field at all sensor locations recorded every time step; this file has prefix "*analytic*" in its name for historical reasons (it might contain data of analytic as well as simulated solution); (2) the file of a number of entire fields with prefix "*true_field*". The latter file contains full (ground-truth) state fields for presentation and comparison against the data assimilation solution. It is important to emphasize that these files can grow up to dozens of gigabytes in case of a large problem. For this reason, we record solution values only at sensor locations on each time step and a quite small number of full fields over the course of simulation to save memory space[1].

5. Input data for C++ `Amdados` application:

   - Text file of sensor locations: "*sensors*.txt*";
   - Text file of *observations* at sensor locations: "*analytic*.txt*".

   The first of these files is generated by `Amdados` application running in special scenario[2], e.g.:

   ```
   build/app/amdados --scenario sensors --config config-file-name
   ```

   For convenience, if a file of sensor locations was not found in the output directory[3] upon launching the script `python/ObservationsGenerator.py`, the `Amdados` application will be automatically run in aforementioned scenario to generate sensors (pseudo) randomly seeded inside the domain.

[1] Parameter *write_num_fields*.
[2] Sensors generation by Python is too slow, so we use C++ implementation instead.
[3] Parameter *output_dir* in configuration file.

6. In order to distinguish simulations with different settings the naming convention was adopted, which is best explained by example:

   `analytic_Nx176_Ny176_Nt1225.txt`,

   where *analytic* is the data type prefix, $N_x = 176$ is the total number of nodal points in x dimension, $N_y = 176$ is the total number of nodal points in y dimension, $N_t = 1225$ is number of time integration step, and *txt/bin* is the extension of a text or binary file respectively. The number of time steps is omitted in the name of sensor locations file because sensors do not change their positions over time.

2.4 Running the Amdados simulation

2.4.1 Single run

1. Step into the project root directory.
2. Create a configuration file using whatever settings you need or just use the default one. It is not recommended to modify the default file `amdados.conf` or tweak any parameter, except for the number of sub-domains in either direction and the integration time, unless you understand all the consequences. Also, note that the size of a sub-domain must coincide with the hard-coded size in C++ code, where it is used for template instantiation. We advice to use `amdados.conf` as a starting point.
3. Build `Amdados` executable in "Release" mode:

   ```
   bash standard.build.sh
   ```

   The building script extracts the output directory from `amdados.conf`. If your preferences are different, please, modify the self-explanatory file `standard.build.sh` accordingly.
4. If you want to get rid of any previous stuff in the output folder it is worth to do this right after building the executable. Note, all the simulation artefacts will go into the output directory.
5. Run the Python[4] script to generate a file of sensor locations (if missed), the file of observations and the file of full-field snapshots:

   ```
   python3 python/ObservationsGenerator.py --config your.conf
   ```

   If not specified, the configuration file `amdados.conf` will be used. The command `--help` lists all the options. Upon completion the following files will be created in the output folder:

   ```
   sensors_Nx*_Ny*.txt
   analytic_Nx*_Ny*_Nt*.txt
   true_field_Nx*_Ny*_Nt*.bin
   ```

[4] We recommend Python 3.6+; Python 3.5+ is the minimum acceptable version.

where the values under symbols "∗" depend on current settings.

6. Run `Amdados` executable in (normal) simulation mode:

```
./build/app/amdados --config your.conf
```

   This will generate the output file `field_Nx*_Ny*_Nt*.bin` containing a number of full-field snapshots of data assimilation solutions (the same number of snapshots evenly distributed over integration period as in aforementioned file of "true" solutions: `true_field_Nx*_Ny*_Nt*.bin`).

7. Putting all together:

```
bash standard.build.sh
/bin/rm -fr output/*          # optional cleanup
python3 python/ObservationsGenerator.py --config amdados.conf
./build/app/amdados --config amdados.conf
```

   The example was shown with the default configuration file `amdados.conf`. When finished, we obtain the snapshots of the "true" and data assimilation solutions that can be compared and visualized, see Section 2.4.4.

   Note, during the last simulation *no console output will be printed*. Please, do not be surprised — this is the project requirement.

   Note, in the meantime data assimilation is slow due to frequent matrix inversion. We recommend to use many-core machine (16+) even for default configuration `amdados.conf`.

   If you need to debug the code, some verbosity as well as additional error checking can be enabled in the file `code/app/include/debugging.h` by un-commenting the line: `#define AMDADOS_DEBUGGING`.

2.4.2 Size scalability test

1. The Python script `python/ScalabilityTestSize.py` runs several simulations with increasing problem size. The doc-string at the beginning of the script provides further details.

2. The script should be launched without any argument provided the `Amdados` application was compiled, for example:

```
bash standard.build.sh
python3 python/ScalabilityTestSize.py
```

3. There are two script's variables that user can modify according to his/her requirements: `GridSizes` and `IntegrationPeriod`. The former is a set of problem sizes (number of sub-domains in both dimensions). Note, the large sizes can result in weeks (if not months) of simulation. As a rule of thumb, a size should not exceed 100×100 sub-domains. Considering the

default size of a sub-domain is 16×16 points, the whole domain size would be 1600×1600 nodal points — quite large problem. The latter variable – `IntegrationPeriod` – gives the integration time in seconds.

4. Besides the aforementioned variables `GridSizes` and `IntegrationPeriod`, which have to be selected by user, everything else is done automatically.

5. Once the test had finished, the results, stored in the output directory, can be visualized, see Section 2.4.4 for further details.

2.4.3 Multi-threading scalability test

1. The Python script `python/ScalabilityTestMT.py` mostly repeats the functionality of the previous one except runs several simulations with the same problem size but increasing the number of CPU cores (threads). The doc-string at the beginning of the script provides further details.

2. The same variables `GridSizes` and `IntegrationPeriod` can be modified by user. No input arguments are required, for example:

```
bash standard.build.sh
python3 python/ScalabilityTestMT.py
```

3. Once the test had finished, the results, stored in the output directory, can be visualized, see Section 2.4.4 for further details.

4. **Important**: we strongly recommend not to use the same size or integration period as in the test `python/ScalabilityTestSize.py`, otherwise some result files can be overwritten with the loss of test information.

2.4.4 Visualization

1. If everything went al-right, we obtain four files in the output folder for a single run, for example:

```
sensors_Nx*_Ny*.txt
analytic_Nx*_Ny*_Nt*.txt
true_field_Nx*_Ny*_Nt*.bin
field_Nx*_Ny*_Nt*.bin
```

   In case of scalability tests a number of quadruples will be generated with different sub-domain numbers (N_x, N_y) and integration time (N_t). Note, the values N_x, N_y and N_t are used as a signature of simulation.

   The symbols "*" hides the actual parameters selected in your configuration.

2. Visualization script `python/Visualize.py` gives an example of how the *binary* files of field snapshots can be read (function `ReadResultFile()`

in the module `python/Utility.py`). Although file of field snapshots is binary, internally it consists of 4 columns (time, abscissa, ordinate, value) of each nodal point. The records are not sorted since Allscale IO API does not guarantees ordering, so we sort the records lexicographically by the triplets (t,x,y) and group them into separate fields (by the time-stamp t).

3. For a single simulation we can run:

   ```
   python3 python/Visualize.py --field_file output/field_Nx*_Ny*_Nt*.bin,
   ```

   where the names of all other files related to the simulation are deduced from the name of data assimilation result.

4. Upon completion one will find the following new files in the output folder:

   ```
   sensors_Nx176_Ny176.png
   rel_diff_Nx176_Ny176.png      ,
   video_Nx176_Ny176_Nt1225.avi
   ```

   where the first image depicts sensor layout and sub-domains, the second picture shows how relative error between the "true" and estimated density changes over the course of time integration, and finally the third video-file demonstrates the evolution of the "true" density field versus data assimilation solution.

5. In case of multiple simulations, it would be easier to run the following bash script for all the results in turn:

   ```
   for f in output/field_*.bin; do
       python3 python/Visualize.py --field_file $f
   done
   ```

   See also the doc-string in the file `python/Visualize.py`. Note, the script should be launched from the project root folder and input file name is prepended by the `output` directory name.