A. Artifact description

This artifact supports the PPoPP'16 paper: Multi-Core On-The-Fly SCC Decomposition.

A.1 Abstract

The artifact described here consists of experiments that evaluate our novel multi-core on-the-fly strongly connected component (SCC) algorithm, called UFSCC [2]. With these experiments, we demonstrate scalability and show that our algorithm outperforms existing work on a 64-core machine on various types of graphs. The current artifact paper describes how to compile and run the necessary tools, and provides a description on how to reproduce all experiments described in the paper [2].

A.2 Description

The goal of the experiments in [2] is to demonstrate the performance improvements with an increasing the number of processors, i.e. the scalability of the algorithm, and compare this performance against existing solutions. The artifact therefore implements the UFSCC algorithm in existing tool sets that perform graph analysis. The artifact comprises two parts:

- Experiments for an *on-the-fly* environment, in which an input graph is provided implicitly and computed on-the-fly. The model checker LTSMIN [4] is used for these experiments. Here, UFSCC is compared with Tarjan's sequential SCC algorithm [10] and Renault's parallel SCC algorithm [8].
- Experiments for an *offline* environment, in which an input graph is provided explicitly, i.e. where all vertices and edges of the graph are known beforehand. The tool provided by Hong et. al [3], which we refer to as HONG-UFSCC, is used for these experiments. Here, UFSCC is compared with Tarjan's sequential SCC algorithm [10] and Hong's parallel SCC algorithm [3].

The structure of the current paper is as follows: We first provide a check list (App. A.2.1) describing specific properties of the artifact. Then, in App. A.2.2, we show where all necessary files are located in the artifact. This is followed by the hardware and software dependencies (App. A.2.3, App. A.2.4). App. A.3 describes the installation procedure. Finally, App. A.4 and App. A.5 explain how to perform and analyze experiments.

A.2.1 Check-list (artifact meta information)

- **Algorithm:** graph, on-the-fly, strongly connected components, multi-core, scalability
- **Program:** LTSMIN model checker, HONG-UFSCC
- **Data set:** included data sets for:
  - LTSMIN: BEEM (BEnchmarks for Explicit Model checkers [7]) + synthetic graphs
  - HONG-UFSCC: real-world + synthetic + selected BEEM graphs
- **Run-time environment:** POSIX / Linux
- **Hardware:** 64bit x86 multi-core machine with 64 cores
- **Output:** CSV with performance info, option to compile PDF table/graph from CSV
- **Experiment workflow:** bash script executes experiments
- **Publicly available?:** open source via Github

A.2.2 How delivered

The source code for the used tools, including the UFSCC implementation, is available at Github:

`https://github.com/utwente-fmt/ppopp16`

This project contains (most of) the experiments combined with scripts to perform them and analyze their results. Structurally, the project contains the following:

- A bash script `install.sh`, this script provides commands to compile and install LTSMIN and HONG-UFSCC (we provide detailed installation instructions in App. A.3).
- A directory `experiments` that contains:
  - A bash script `benchmark.sh`, this script is used to run all experiments.
  - A Python script `parse-output.py`, this script is used to analyze the program output and store the results in a CSV file (or in case of a failure, copy the program output to a failure folder).
  - A Python script `csv2graph.py`, this script generates either a time or speedup graph for a specific graph.
  - A Python script `csv2table.py`, this script generates a table containing info regarding times and speedups for multiple graphs.
  - A Python script `csv2scatter.py`, this script generates a scatterplot containing info regarding times and speedups for multiple graphs.
  - A bash script `parse-csv.sh`, this script is intended as an example on how to extract graphs/tables from the generated CSV data (using the `csv2..` scripts).
  - A directory `results-paper` that contains the CSV outputs for our experiments that we present in the PPoPP paper.
  - A directory `graphs`, that contains:
    - A directory `beem`, this contains a set of input graphs obtained from the BEEM database.
    - A directory `beem-selected`, this contains the input graphs for six 'selected' input graphs from the beem directory.
    - A directory `synthetic`, this contains several synthetically generated input graphs.

Due to size restrictions, the input graphs for the 'offline' experiments are provided in a separate zip file, provided at:

`http://mab.to/MHjH2OC90`

This zip file contains two non-empty directories: `real-offline` and `constructed-offline`. In order to perform the offline experiments, please copy these directories to the `graphs` directory.

The `real-offline` directory contains various real-world graphs. The `livej` [1], `patent` [11] and `pokec` [9] graphs were obtained from the SNAP [5] database. The graph `wiki-links`[1] represents Wikipedia's page-to-page links. The `random`, `rmat` and `ssca2` graphs represent random graphs with real-world characteristics and were generated from the GTGraph [6] suite using default parameters. Using the Green-Marl tool [12], the graphs were transformed to a binary format (to be used in the HONG-UFSCC tool).

A.2.3 Hardware dependencies

To replicate our experiments, a 64-core x86 machine with 64 bit addressing is required. All experiments in [2] were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of 64 cores and 128GB of main memory.

The experiments are specifically (hardcoded) set up to be performed with 64 cores. It is still possible to run the scripts with fewer

[1] Obtained from `http://haselgrove.id.au/wikipedia.htm`

cores available, although the results for benchmark runs with more threads than the available cores should be discounted. Benchmarking with more than 64 cores is not possible due to limitations of the tools.

A.2.4 Software dependencies

We recommend evaluating the artifact on POSIX / Linux. We performed all experiments on Ubuntu 14.04. To install LTSMIN and HONG-UFSCC and run the experiments, the following dependencies are required:

- `g++` (with builtin atomic functions and OpenMP support)
- `popt` (version $\geq$ 1.7, package `libpopt-dev`)
- `zlib` (package `zlibc zlib1g zlib1g-dev`)
- `Flex` (package `flex`)
- `Apache Ant` (package `ant`)
- `make`
- `automake`
- `autoconf`
- `libtool`
- `DiVinE` (patched, see App. A.3.1 for installation)
- `Python` 2.7.X with the modules `numpy` and `scipy` (packages `python-numpy` and `python-scipy`)
- A `Latex` installation with the `pgfplots` package installed (package `texlive-latex-extra`)

A.3 Installation

The current section describes how to install the LTSMIN tool in App. A.3.1 and how to install the HONG-UFSCC tool in App. A.3.2.

A.3.1 Installing LTSmin

Before installing LTSMIN, a patched version of DiVinE 2.4 is required. The source code for DiVinE is available at:

`https://github.com/utwente-fmt/divine2.git`

In the `divine2` directory, use the following commands to install DiVinE:

- `mkdir _build && cd _build`
- `cmake .. -DGUI=OFF -DRX_PATH= -DCMAKE_INSTALL_PREFIX=install -DMURPHI=OFF`
- `make`
- `make install`

(see `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`) Here, `install` is a directory (included in the PATH) where to install DiVinE.

The source code for LTSMIN is available at:

`https://github.com/vbloemen/ltsmin/tree/vincent3`

In the `ltsmin` directory, use the following commands to install LTSMIN:

- `git submodule update --init`
- `./ltsminreconf`
- `./configure --prefix=install`
- `make && make install`

(see also `http://fmt.cs.utwente.nl/tools/ltsmin/`) Here, `install` is a directory (included in the PATH) where to install LTSMIN. To test whether LTSMIN has been installed correctly, in the `experiments/graphs` directory, use the command `dve2lts-mc --strategy=ufscc -s28 beem/adding.3.dve`. This runs the UFSCC algorithm on the `adding.3` graph. If successful, various performance statistics will be printed on the standard output (for instance `total scc count: 1894376`).

A.3.2 Installing Hong-UFSCC

The source code for the HONG-UFSCC tool is available at Github:

`https://github.com/vbloemen/hong-ufscc.git`

In the `hong-ufscc` directory, the code is compiled with the command: `make`. If successful, the current directory now contains an executable: `scc`. Calling `scc` with no parameters will print the help message, which describes the options. To make this executable usable for the experiments, please make sure that the directory containing the `scc` executable is included in the PATH.

A.4 Experiment workflow

The experiments can be executed with the provided bash script from the Github project (`bench.sh`).

The procedure to perform all experiments from the paper is as follows:

1. Install LTSMIN and HONG-UFSCC (see App. A.3).
2. Copy the offline input graphs from the zip file to the directories:
   - `experiments/graphs/real-offline`
   - `experiments/graphs/constructed-offline`
3. `cd experiments`
4. Run `./bench.sh` to perform all experiments,

The `bench.sh` script will iterate over all files in the subdirectories of `experiments/graphs` and perform a sequence of experiments for each of these graphs. For each experiment, the program is called (LTSMIN or HONG-UFSCC) with as arguments the number of processor instances, the algorithm to be used, and the input graph. The output of the program is stored in a temporary file and analyzed by `experiments/parse-output.py`. This Python script will then parse the output file and append the results in the corresponding CSV file in the `experiments/results` directory.

For the results in the paper, we ran all experiments 50 times. The `bench.sh` script is set up to run the experiments 10 times (which should be a sufficient number to obtain meaningful results).

Specific experiments: The `bench.sh` script was written so that it can be easily modified to perform an experiment on a single graph, using a algorithm, using a specific number of cores, or combinations of these aspects.

A.5 Evaluation and expected result

After performing the experiments, the CSV files in the directory `experiments/results` should contain performance information for each experiment (one row per experiment). We provide three Python scripts to translate the CSV data to Latex files, which can consecutively be compiled to PDF files. The `parse-csv.sh` script provides examples on how to use these Python scripts. We show how to use these scripts as follows:

- `python csv2graph.py INFILE MODEL`, this generates the Latex code to create a graph that depicts how the performance times for the algorithms change when increasing the number of cores, on the graph MODEL obtained from the CSV file INFILE.
- `python csv2graph.py INFILE MODEL ALG N` this generates the Latex code to create a graph that depicts the relative

performance (speedup) of the algorithms compared to a baseline (algorithm ALG, using N cores), on the graph MODEL obtained from the CSV file INFILE.

- `python csv2table.py INFILE` this generates the Latex code to create a table that depicts the performance times for the algorithms (using 64 cores) and the relative performance of UFSCC compared to the others, on all graphs from the CSV file INFILE.
- `python csv2scatter.py INFILE ALG1 WORKERS1 ALG2 WORKERS2`, this generates the Latex code to create a scatterplot that compares the relative performance of algorithm ALG1 using WORKERS1 cores with ALG2 using WORKERS2 cores.

Expected results: The obtained results for all experiments should be similar to those that are provided in the directory `experiments/results-paper`. We expect similar *relative* performance of the algorithms, i.e. if we observe from the results `experiments/results-paper` that algorithm A is 5 times faster than algorithm B on graph X using Y cores, then this same 5-fold time-improvement should also be visible in the results from `experiments/results` (using the same configuration). We assume that the absolute performance times are too dependent on the specifics of the used hardware for valid comparison. The suggested method to make this comparison is by running `csv2table.py` and `csv2scatter.py` on each CSV file in the `experiments/results` and `experiments/results-paper` directories. We expect to observe that:

- The speedup graphs for every input graph should look similar to those from the `results-paper` directory.
- On `beem` graphs, UFSCC (using 64 cores) is typically 10 to 30 times faster compared Tarjan's algorithm
- On `beem` graphs, UFSCC (using 64 cores) is compared to Renault's algorithm (using 64 cores) at worst 0.7 times as fast and for half of the graph instances UFSCC is more than 10 times faster. We consider this result to be the main selling point for our algorithm.
- On the `constructed-offline` graphs that correspond the `beem-selected` ones, UFSCC is expected to perform slightly outperform Hong 's algorithm. and Tarjan's algorithm. Also, Tarjan's algorithm is expected to outperform UFSCC for most of the graphs.
- On the `real-offline` graphs, UFSCC outperforms Tarjan's algorithm, but Hong's algorithm outperforms UFSCC (except for `wiki-links`).
- We expect that some failed experiments, especially for experiments on `constructed-offline` graphs using Hong's algorithm. We have also observed a few failures for Renault's algorithm. We refer to the contents of the `experiments/failures` directory for the failed experiments.

References

[1] Backstrom, et al. (2006). Group Formation in Large Social Networks: Membership, Growth, and Evolution. KDD.

[2] Bloemen, V., Laarman, A. W., & van de Pol, J. C. (2016). Multi-Core On-The-Fly SCC Decomposition. In PPoPP 2016. IEEE.

[3] Hong, S., Rodia, N. C., & Olukotun, K. (2013). On fast parallel detection of strongly connected components (SCC) in small-world graphs. In High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for (pp. 1-11). IEEE.

[4] Kant et al. (2015). LTSmin: High-Performance Language-Independent Model Checking. In TACAS.

[5] Leskovec, J. SNAP: Stanford network analysis project. http://snap.stanford.edu/index.html, last accessed 9 Sep 2015.

[6] Madduri, K., & Bader, D. A. GTgraph: A suite of synthetic graph generators. http://www.cse.psu.edu/ kxm85/software/GTgraph/, last accessed 9 Sep 2015.

[7] Pelánek, R. (2007). BEEM: Benchmarks for Explicit Model Checkers. In SPIN, volume 4595 of LNCS, pp. 263–267. Springer

[8] Renault, E. et al. (2015). Parallel explicit model checking for generalized Büchi automata. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 613-627). Springer Berlin Heidelberg.

[9] Takac, L., & Zábovský, M. Data Analysis in Public Social Networks, International Scientific Conference & International Workshop Present Day Trends of Innovations, May 2012 Lomza, Poland.

[10] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. SIAM. Comput. 1:146-60.

[11] Leskovec, J., Kleinberg, J., & Faloutsos, C. (2005). Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).

[12] S. Hong et al, "Green-Marl: A DSL for easy and efficient graph analysis", ASPLOS 2012