SOFTWARE METAPAPER

PerfAndPubTools – Tools for Software Performance Analysis and Publishing of Results

Nuno Fachada¹, Vitor V. Lopes², Rui C. Martins³ and Agostinho C. Rosa¹

¹ Institute for Systems and Robotics, LARSyS, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal nfachada@laseeb.org
² Universidad de las Fuerzas Armadas-ESPE, Sangolquí, Ecuador
³ Life and Health Sciences Research Institute, School of Health Sciences, University of Minho, Braga, Portugal

Corresponding author: Nuno Fachada

PerfAndPubTools consists of a set of MATLAB/Octave functions for the post-processing and analysis of software performance benchmark data and producing associated publication quality materials.

Keywords: Profiling; Benchmarking; Performance analysis; Publishing tools

Funding statement: This work was supported by the Fundação para a Ciência e a Tecnologia (FCT) projects UID/EEA/50009/2013 and UID/MAT/04561/2013, and partially funded with grant SFRH/BD/48310/2008, also from FCT. The author Vitor V. Lopes acknowledges the financial support from the Prometeo project of SENESCYT (Ecuador).

(1) Overview

Introduction

PerfAndPubTools consists of a set of MATLAB [1], GNU Octave-compatible [2], functions for the post-processing and analysis of software performance benchmark data and producing associated publication quality materials. More specifically, the functions bundled with PerfAndPubTools allow to:

1. Batch process files containing benchmarking data of computer programs, one file per run.
2. Determine the mean and standard deviation of benchmarking experiments with several runs.
3. Organize the benchmark statistics by program implementation and program setup.
4. Output scalability and speedup data, optionally generating associated figures.
5. Create publication ready benchmark comparison tables in LATEX.

These tools were originally developed to assess the performance of serial and parallel implementations of the PPHPC simulation model [3], as well as for producing some of the associated publication quality materials. However, the tools can be used with any computational benchmark experiment.

Implementation and architecture

Performance analysis in PerfAndPubTools takes place at two levels: implementation and setup. The implementation level is meant to be associated with specific software implementations for performing a given task, for example a particular sorting algorithm or a simulation model realized in a certain programming language. Within the context of each implementation, the software can be executed under different setups. These can be different computational sizes (e.g., vector lengths in a sorting algorithm) or distinct execution parameters (e.g., number of threads used).

PerfAndPubTools is implemented in a layered architecture using a procedural programming approach, as shown in Figure 1. From lowest to highest-level of functionality, the functions represented in this Figure have the following roles:

![Figure 1: PerfAndPubTools architecture. Blocks in typewriter font represent functions. Dashed blocks represent directly replaceable functions.](image-url)
get_gtime: Given a file containing the default output of the GNU time [4] command, this function extracts the user, system and elapsed times in seconds, as well as the percentage of CPU usage.

gather_times: Loads execution times from files in a given folder. This function uses get_gtime by default, but can be configured to use another function to load individual benchmark files with a different format.

perfstats: Determines mean times and respective standard deviations of a computational experiment, optionally plotting a scalability graph if different setups correspond to different computational work sizes.

speedup: Determines the average, maximum and minimum speedups against one or more reference implementations across a number of setups. Can optionally generate a bar plot displaying the various speedups.

times_table: Returns a matrix with useful contents for using in tables for publication, namely times (in seconds), absolute standard deviations (seconds), relative standard deviations, and speedups against one or more reference implementations.

Implementation specs can be defined in a similar fashion. Note that all implementation specs must have the same number of setups, and corresponding setups should have the same sname. Additionally, plotting with perfstats requires that the computational size, csize, is defined and has the same value for corresponding setups in different implementations specs.

Algorithm scalability: The perfstats function determines mean times and standard deviations of individual setups for each implementation. If the various setups correspond to different computational work sizes, perfstats can optionally plot a scalability graph. The following instruction performs this task for the experimental setup under discussion:

```
[m, s] = perfstats(3, 'Bubble', bs, ...
    'Selection', ss, 'Merge', ms, ...
    'Quick', qs);
```

The contents of the returned variables, m and s, are as follows:

<table>
<thead>
<tr>
<th>m</th>
<th>36.0040</th>
<th>144.8210</th>
<th>325.1730</th>
<th>577.8600</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9.5270</td>
<td>38.0500</td>
<td>88.5130</td>
<td>153.6560</td>
</tr>
<tr>
<td></td>
<td>0.0200</td>
<td>0.0410</td>
<td>0.0600</td>
<td>0.0850</td>
</tr>
<tr>
<td></td>
<td>0.0100</td>
<td>0.0200</td>
<td>0.0300</td>
<td>0.0510</td>
</tr>
<tr>
<td>s</td>
<td>0.8873</td>
<td>2.9223</td>
<td>6.1874</td>
<td>6.3846</td>
</tr>
<tr>
<td></td>
<td>0.0690</td>
<td>0.2829</td>
<td>3.6976</td>
<td>3.0600</td>
</tr>
<tr>
<td></td>
<td>0.0000</td>
<td>0.0032</td>
<td>0.0000</td>
<td>0.0127</td>
</tr>
<tr>
<td></td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0032</td>
</tr>
</tbody>
</table>

The m variable represents means times (in seconds), while s holds the respective standard deviations. Rows are associated with implementations (i.e., sorting algorithms), while columns represent setups (i.e., vector sizes).

The first parameter of perfstats specifies whether to generate a scalability plot. More specifically, the value 3 orders the function to generate a semi-logarithmic plot,
as shown in Figure 2. Negative values indicate that the
figure should also display error bars representing the
standard deviation in the measured computational sizes.
No plot will be generated if zero is passed as the first
argument.

Obtaining the speedup: The speedup function deter-
mines speedups against one or more reference implementa-
tions, across a number of setups. Its usage is similar to
that of perfstats, requiring the identification of the
implementation specs to compare:

```matlab
[s_avg, s_max, s_min] = speedup(-2, 1, ...
    'Bubble', bs, 'Selection', ss, ...
    'Merge', ms, 'Quick', qs);
```

The first parameter concerns the optional bar plot the
function is able to generate. An absolute value of 2 states
that a bar plot with a logarithmic scale should be generated,
as shown in Figure 3. Since this value is negative, error
bars representing the maximum and minimum speedups
are drawn on top of the average speedup bars. The second
parameter defines the reference implementation(s) to
which the speedups are to be determined against. Passing
1 identifies the first implementation, Bubble sort, as the
reference. The speedup function returns cell arrays con-
taining the average, maximum and minimum speedup
matrices for each reference implementation. In this case,
one reference was defined, and thus only the first item in
the returned cells is available:

```
>> s_avg(1) = 1.0e+04 *
    0.0001   0.0001   0.0001   0.0001
    0.0004   0.0004   0.0004   0.0004
    0.1800   0.3532   0.5420   0.6798
    0.3600   0.7241   1.0839   1.1331
```

In a similar fashion to the mean and standard deviation
matrices returned by perfstats, rows of speedup matrices
are associated with implementations (i.e., sorting
algorithms), while columns represent setups (i.e., vector
sizes). Note that, in this case, the first row represents the
average speedup of Bubble sort against itself, and, as such,
the values are all ones.

Generating tables: PerfAndPubTools can generate
plain text or publication quality tables summarizing
the performed computational benchmarks. The process
is divided in two steps using the times_table and
times_table_f functions, respectively. The former
determines and returns a matrix containing partial or
complete information to generate a table, while the latter
effectively generates tables. This division is useful because
times_table_f can accept more than one matrix
returned by times_table, allowing the generation of
more complex tables.
The `times_table` function, like `perfstats` and `speedup`, requires the identification of the implementation specs to compare, as shown in the following command:

```matlab
  tdata = times_table(1, ...
    'Bubble', bs, 'Selection', ss, ...
    'Merge', ms, 'Quick', qs);
```

The first argument designates the references implementation or implementations, in a similar fashion to the second parameter of `speedup`. The return value, `tdata`, can be passed to `times_table_f` in order to generate a table:

```matlab
  times_table_f(0, 'vs Bubble', tdata)
```

The first argument, 0, instructs the function to generate a plain text table, as shown in Figure 4. Setting this value to 1 would generate a LATEX table, as shown in Figure 5. Note that LATEX tables require the `siunitx`, `multirow`, and `booktabs` packages.

**Complete example**: The complete example is available in the user manual bundled with the software. It contains the necessary steps required to reproduce these results, also showing how the return values of `perfstats` and `speedup` can be used to generate custom publication quality plots. The user manual also details an additional example concerning the performance of serial and parallel implementations of the PPHPC simulation model [3], namely different ways of contextualizing the concept of computational size, and the generation of more complex tables.

**Quality control**
The available functions are covered by unit tests in order to ensure their correct behavior. The MOxUnit framework [11] is required for running the unit tests. Additionally, all the examples available in the user manual (bundled with the software) have been tested in both MATLAB and Octave.

(2) **Availability**

- **Operating system**
  Any system capable of running MATLAB R2013a or GNU Octave 3.8.1, or higher.

- **Programming language**
  MATLAB R2013a or GNU Octave 3.8.1, or higher.

- **Dependencies**
  There are no additional dependencies for the package tools. However, unit tests depend on the MOxUnit unit test framework for MATLAB and GNU Octave.
These utilities can be used for analyzing any computational experiment. As described in ‘Implementation and architecture’, other benchmark data formats can be specified by implementing a custom function to replace `get_gtime` and setting its handle in the `gather_times` function. Results from `perfstats` and `speedup` functions can be used to generate other types of figures. The same is true for `times_table`, the results of which can be integrated in table layouts other than the one provided by `times_table_f`.

### Competing Interests

The authors declare that they have no competing interests.

### Acknowledgements

This software is enhanced by the `matlab2tikz` script and by the `siunitx`, `multirow` and `booktabs` LATEX packages.
Note 1 e.g., files containing the output of GNU time.

References

Submitted: 26 January 2016 Accepted: 26 April 2016 Published: 12 May 2016

Copyright: © 2016 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See http://creativecommons.org/licenses/by/4.0/.