Proceedings of the 2nd International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/132/2024.20797

Performance Shell Benchmark Correctness, Efficiency, and Beyond

Yizhang Xu
Computer Science, Beijing University of Posts and Telecommunications, Beijing, China

xuyizhang@bupt.edu.cn

Abstract. Shell script is pivotal in tackling diverse real-world issues such as COVID-19 analytics, NLP (Natural Language Processing), and data analysis. Yet, they can be prone to failure or inefficiency. This paper rigorously assesses the reliability and speed of these benchmark shell scripts. For verifying output integrity, this study leverage SHA-256 hashes and the diff utility for swift and accurate comparisons with the correct outputs. To measure performance, the time command is employed to capture and log execution times. Ultimately, our analysis reveals that 82/204 of the shell script outputs are accurate, demonstrating robust performance even under the demands of large-scale data processing.

Keywords: Shell Script, PaSh, Reliability and Speed, Performance Measurement, SHA-256 hashes

1. Introduction

Shell[1] is a command-line interface for interacting with the operating system[2]. In the dynamic landscape of computer science, shell scripts are indispensable tools that frequently engage with complex text problems, playing a pivotal role in the functionality of computer systems[3]. It is not uncommon for a system to house hundreds of these scripts, each a thread in the tapestry of system operations[4][5]. By leveraging parallelization systems like POSH[6], PaSh[7], and DiSh[8], among others, computers can significantly enhance script execution speed.

The core of the issue lies in the imperative to ensure the scripts' accuracy and performance. A deviation from the intended path or a substandard execution mode can lead to inefficiencies and, more critically, to erroneous outcomes that undermine the integrity of the system's operations[5].

To address these concerns, this paper embarks on a meticulous examination of shell scripts across four repositories, including covid-mts, oneliners, nlp[9] and file-enc. Our approach involves a blend of testing, debugging, and the provision of strategic advice aimed at optimizing the scripts' performance.

Our findings are both revealing and instructive: out of 204 shell scripts tests, 82 successfully delivered correct results. However, one script's performance stood out, taking over 5 minutes to handle a 1GB text file, underscoring the necessity for performance enhancements, especially when dealing with voluminous data sets.

2. Related Work

This study focuses on evaluating and optimizing the performance of benchmark scripts[10], particularly their efficiency and accuracy when processing large-scale datasets.

DOI: 10.54254/2755-2721/132/2024.20797

PaSh[7] is a system for parallelizing shell scripts, and it generates benchmark scripts used for testing in this study. PaSh highlights the widespread use of POSIX shell scripts in system management, automation, and data processing, but also points out the inefficiency of their typically sequential execution.

PaSh-JIT[11] focuses on the real-time aspects of script execution. PaSh-JIT dynamically interposes parallelization during the execution of shell scripts. It ensures that the performance enhancements achieved through parallelization do not compromise the script's original functionality or correctness. This approach aligns with our methodological emphasis on maintaining reliability while enhancing script performance.

DiffStream[12] is a system designed to facilitate differential testing for stream processing programs. DiffStream compares the outputs of different versions of a program or different programs running on the same data streams, highlighting any inconsistencies between them. This study follows DiffStream's method and evaluate PaSh benchmark.

This study bears similarities to the work conducted by the KISTI team [13]. They also established a standard dataset for assessing and comparing tool performance. However, our study emphasizes simplifying the installation, execution, and data collection of benchmark suites through an automated process, especially on the simulated the benchmark shell scripts.

The project "NVIDIA Performance Testing for Emulation of the Grace CPU"[14], is directly related to this study. They developed a novel tool that consolidates other test suites for more convenient testing of NVIDIA's simulated CPU. Building upon their work, this study further explores how to enhance benchmark efficiency and maintainability through script optimization and the use of high-level programming languages.

3. Example of Benchmark Test

An Example: Take **oneliners** as an example of our evaluation. To evaluate the execution time of scripts categorized under **oneliners**, you can follow these steps:

3.1. Inputs:

```
./inputs.sh
```

In the inputs.sh script, data is downloaded into the designated directory named inputs.

3.2. Run

```
./run.sh --small
```

The run.sh script is responsible for executing all shell programs located in the scripts folder. It also logs the time taken for each execution and appends this information to a file named **oneliners**.res within the output folder. The "–small" option, when specified, indicates that the shell should be run with a smaller dataset to expedite the execution process. Ideally, repeat this step five times and take the average as the final time.

3.3. Verify

```
./verify.sh --small
```

The verify.sh script performs a check on the correctness of the output generated by the shell programs in the scripts folder. When the "–small" option is applied, it ensures that the shell program outputs are accurate even when run with the reduced dataset.

DOI: 10.54254/2755-2721/132/2024.20797

3.4. Clear

```
./cleanup.sh
```

The cleanup.sh script is designed to perform a cleanup operation by removing downloaded files and generated outputs, thereby freeing up system memory.

3.5. Results

The results of these operations, including timing and verification outcomes, are compiled and recorded in the oneliners.res file.

```
executing oneliners bash 08.05.2024 Monday 15:40:23 CST ./scripts/nfa-regex.sh
2.460
./scripts/sort.sh 0.076
./scripts/top-n.sh 0.309
./scripts/wf.sh 0.285
./scripts/spell.sh 0.529
./scripts/diff.sh 0.138
./scripts/bi-grams.sh 0.392
./scripts/set-diff.sh 0.187
./scripts/sort-sort.sh 0.126
./scripts/shortest-scripts.sh 0.698
```

4. Background

This paper depends on the following techniques and shell scripts code.

4.1. PaSh

PaSh[7] is an innovative system designed for the parallelization of POSIX shell scripts. At its core, PaSh automatically transforms scripts into a dataflow graph, revealing potential parallel execution paths through a series of semantic-preserving transformations. It then recompiles the optimized dataflow graph back into a script, incorporating specific POSIX directives and PaSh's proprietary runtime primitives to ensure the efficiency and correctness of parallel execution. PaSh offers a concise annotation language that allows command developers to easily mark the parallel characteristics of their commands. Tested across multiple Unix scripts, PaSh has demonstrated significant performance improvements, affirming its effectiveness in automating the parallelization of scripts.

4.2. Shell Script Benchmarks

Shell script benchmark[10] repository under binpash on GitHub is a curated collection of 16 benchmark programs and 105 shell scripts enhanced with PaSh. This resource provides a robust platform for assessing the impact of parallelization on shell script execution. It offers a diverse range of scripts, from simple to complex, enabling a comprehensive evaluation of PaSh's ability to accelerate shell-based workflows. The repository is a valuable tool for developers and researchers working in parallel computing within the shell scripting domain.

5. Implementation

To achieve the desired outcome, adhere to the following sequential steps:

5.1. Linux Environment and PaSh

Virtual Machine Installation: On your Windows or macOS system, begin by installing a virtual machine software such as VMware. Proceed to set up an Ubuntu image file within this virtual

DOI: 10.54254/2755-2721/132/2024.20797

environment.

Installing PaSh: Install the PaSh by executing the following command in terminal[7]:

```
wget https://raw.githubusercontent.com/binpash/pash/main/scripts
/up.sh
sh up.sh
export PASH_TOP="$PWD/pash/"
## Run PaSh with echo hi
"$PASH_TOP/pa.sh" -c "echo hi"
```

5.2. Benchmark

Install the binpash/benchmark repository by executing the following command in terminal:

```
git clone https://github.com/binpash/benchmarks.git
```

For the vast majority of projects, run by imitating the Example section. If there are special requirements, refer to the README file.

6. Evaluation

To conduct a comprehensive benchmark evaluation, we have selected four packages from the benchmark suite and aim to address the following inquiries:

Q1: Do the shell scripts produce the correct outcomes?(5.1)

Q2: How does the shell script perform when processing extensive datasets?(5.2)

The run.sh executes all scripts and record the running time. The verify.sh compare the hash values of the shell script outputs with the hash values of the expected answers[15]. If the scripts complete their tasks in under 600 seconds when dealing with large-scale data, they are deemed efficient. The congruence of these hashes will confirm the correctness of the shell script's response.

6.1. Correctness

Overall, 40.2% (82/204) of the outputs matched the benchmark hashes. The detailed breakdown by category is as follows:

- covid-mts: 0% (0/4) of outputs matched.
- nlp: 41.1% (78/190) of outputs matched.
- oneliners: 40% (4/10) of outputs matched.

The scripts nfa-regex.sh, spell.sh, top-n.sh, and wf.sh produced correct outputs. However, the verify.sh script within the file-enc suite was empty, and the file-enc directory did not include the necessary folder hashes. Due to these omissions, the paper is unable to verify the correctness of the shell scripts in the file-enc.

6.2. Efficiency

Covid-mts: The execution time of the scripts in the covid-mts are detailed in Table 1. All the scripts completed their task in 300s when dealing with a 3.4GB file. The 3.sh cost the most of time, taking 80.081s on average. The 1.sh and 2.sh cost neerly the same, separately using 66.137s and 66.069s. The 4.sh cost least, using 25.581s.

Oneliners: The execution times of the scripts in oneliners are detailed in Table 2. All scripts completed their tasks in under 300s when processing 334MB of data. Among them, the nfa-regex.sh script demonstrated the highest latency, with an average runtime of 126.397s. The average execution time of bi-grams.sh is 66.214. The execution time of wf.sh, top-n.sh, spell.sh is around 45s, separately 45.137s, 45.090s, 43.705s. The average execution time of set-diff.sh is 34.588s. The execution time of

DOI: 10.54254/2755-2721/132/2024.20797

Table 1: running time of covid-mts scripts

covid-mts	AVG/s	MAX time/s	Min time/s
1.sh	66.137	66.589	65.766
2.sh	66.069	66.383	65.747
3.sh	80.081	80.725	79.639
4.sh	25.581	25.728	25.462

Table 2: running time of oneliners scripts

oneliners	AVG/s	MAX time/s	Min time/s
nfa-regex.sh	126.397	127.222	126.010
bi-grams.sh	66.214	67.416	65.693
wf.sh	45.137	45.225	45.006
top-n.sh	45.090	45.158	45.038
spell.sh	43.705	44.169	43.531
set-diff.sh	34.588	34.840	34.472
sort-sort.sh	19.489	19.576	19.436
shortest-scripts.sh	18.005	18.045	17.947
diff.sh	16.189	16.312	16.050
sort.sh	13.330	13.354	13.275

sort-sort.sh, shortest-scripts.sh and diff.sh is nearly the same, separately 19.489s, 18.005s and 16.189. The sort.sh cost the least, whose average execution time is 13.330s. The relatively high runtime of nfa-regex.sh can be attributed to its need to match complex regular expressions over the input data, which is inherently a more resource-intensive task.

Nlp: The execution times of the scripts in oneliners are detailed in Table 3. All scripts completed their tasks in under 300s when processing totally 71MB of data. The 8_3.3.sh runs the longest time, on average 44.605s. Then execution times of the 4_3b.sh, 4_3.sh, 8.2_2.sh, 3_3.sh, 6_2.sh and 8.3_2.share are around 30s, separately 32.622s, 31.419s, 30.545s, 30.370s, 28.735s, 27.188s. Then the execution times of 3_2.sh, 8_1.sh, 1_1.sh, 6_1_2.sh, 2_1.sh, 8.2_1.sh are around 20s, separately 24.198s, 23.375s, 21.442s, 20.803s,19.857s, 19.676s. Then the execution times of 7_2.sh and 6_4.sh are nearly the same, separately 16.928s and 13.906s. Next, the execution times of 6_5.sh, 6_3.sh and 6_1_1.sh are around 9s, separately 9.415s, 9.008s and 8.359s. Then the execution times of 7_1.sh, 6_1.sh, 2_2.sh, 3_1.sh, and 6_7.sh are under 5s, separately 4.780s, 4.458s, 3.407s, 2.921s, 1.924s.

File-enc: The execution times of the scripts in file-enc are detailed in Table 4. The compress_files.sh runs 443.042s on average, whose execution exceeds 300 seconds. The encrypt_files.sh runs 84.374s on average.

The compress_files.sh compress the .png file using gzip command. The execution time of gzip depends on the compression level. The encrypt_files.sh transfrom the file into another file via CBC encrypt mode.

7. Discussion

The file-enc has some bug. The first one is that in run.sh input_dir is wrong and it needs go to the folder inputs. Change like this can solve this bug.

```
# in run.sh wrong in line 9 and 12
input_dir="inputs/pcap_data_small" # should go to inputs
```

The second is that in compress_files.sh and in encrypt_files.sh and miss a folder too.

DOI: 10.54254/2755-2721/132/2024.20797

Table 3: running time of nlp scripts

nlp	AVG/s	MAX time/s	Min time/s
1_1.sh	21.442	21.787	21.148
2_1.sh	19.857	20.050	19.664
2_2.sh	3.407	3.465	3.345
3_1.sh	2.921	23.533	22.577
3_2.sh	24.198	24.366	24.073
3_3.sh	30.370	30.588	20.080
4_3.sh	30.545	31.124	29.963
4_3b.sh	32.622	33.702	31.876
6_1.sh	4.458	4.693	4.371
6_1_1.sh	8.359	8.668	8.128
6_1_2.sh	20.803	21.029	20.608
6_2.sh	28.735	29.131	28.418
6_3.sh	9.008	9.128	8.807
6_4.sh	13.906	14.187	13.677
6_5.sh	9.415	9.697	9.211
6_7.sh	1.924	1.980	1.879
7_1.sh	4.780	4.814	4.672
7_2.sh	16.928	17.180	16.788
8_1.sh	23.375	23.685	22.993
8.2_1.sh	19.676	19.810	19.500
8.2_2.sh	31.419	31.749	30.961
8.3_2.sh	27.188	27.937	26.897
8_3.3.sh	44.605	45.207	44.024

Table 4: running time of file-enc scripts

file-enc	AVG/s	MAX time/s	Min time/s
compress_files.sh	443.042	443.293	442.672
encrypt_files.sh	84.374	84.676	84.068

```
# in compress_files.sh, wrong in line 8
cat $1/$item | gzip -1 -c > $2/$output_name #miss $1

# in encrypt_files.sh, wrong in line 17
cat $1/$item | pure_func > $2/$output_name # miss $1 too
```

After add the folder, scripts can run successfully.

8. Conclusion

The paper test several repositories in benchmark and evaluate the correctness and efficiency. Most of the shell scripts may output in a wrong format which cause the low correct rate. Most execution time of shell scripts is less than 300s when processing in the large-scale data, showing efficiency of these shell.

All the data are open source and available for download:
https://github.com/yiz853793/benchmark-test.

References

[1] Chris Johnson. *Shell Scripting Recipes: A Problem-solution Approach*. Apress, 2006.
[2] Cameron Newham. *Learning the bash shell: Unix shell programming*. ” O’Reilly Media, Inc.”, 2005.

DOI: 10.54254/2755-2721/132/2024.20797

[3] Ganesh Sanjiv Naik. *Learning Linux Shell Scripting: Leverage the power of shell scripts to solve real-world problems*. Packt Publishing Ltd, 2018.

[4] Leonardo Leite, Carlos Eduardo Moreira, Daniel Cordeiro, Marco Aurélio Gerosa, and Fabio Kon. Deploying large-scale service compositions on the cloud with the choreos enactment engine. In *2014 IEEE 13th international symposium on network computing and applications*, pages 121–128. IEEE, 2014.

[5] Andre Goforth. The role and impact of software coding standards on system integrity. In *AIAA Infotech@ Aerospace (I@ A) Conference*, page 5222, 2013.

[6] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. {POSH}: A {Data-Aware} shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[7] Nikos Vasilakis and Konstantinos Kallas. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 49–66, 2021.

[8] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. {DiSh}: Dynamic {Shell-Script} distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, 2023.

[9] Dan Ofer, Nadav Brandes, and Michal Linial. The language of proteins: Nlp, machine learning & protein sequences. *Computational and Structural Biotechnology Journal*, 19:1750–1758, 2021.

[10] Dimitris Karnikis and Tammam Mustafa. Binpash benchmark.

[11] Tammam Mustafa. *Parallel and Distributed Just-in-Time Shell Script Compilation*. PhD thesis, Massachusetts Institute of Technology, 2022.

[12] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[13] Jinsuk Kim, Dong-Hoon Yoo, Heejin Jang, and Kimoon Jeong. Webshark 1.0: a benchmark collection for malicious web shell detection. *Journal of Information Processing Systems*, 11(2):229–238, 2015.

[14] Gong Fan. *NVIDIA Performance Testing for Emulation of the Grace CPU*. PhD thesis, NVIDIA Corporation, 2021.

[15] Olga Manankova, Mubarak Yakubova, and Alimjan Baikenov. Cryptanalysis the sha-256 hash function using rainbow tables. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)*, 10(4):930–944, 2022.