POPL: G: Data-Parallel Shell Scripting

KONSTANTINOS KALLAS, University of Pennsylvania

This paper introduces PaSh, a new shell that exposes data parallelism in POSIX shell scripts. To achieve that, PaSh proposes: (i) an order-aware dataflow model that captures a fragment of the shell, (ii) a set of dataflow transformations that extract parallelism and have been proven to be correct, (iii) a lightweight framework that captures the correspondence of shell commands and order-aware dataflow nodes, and (iv) a just-in-time compilation framework that allows for effective compilation despite the dynamic nature of the shell. A preliminary evaluation of PaSh on 50 unmodified Unix scripts on a 16-core machine shows significant speedups (up to 14.93×, avg: 4.15×).

1 INTRODUCTION

The Unix shell is an attractive choice for specifying succinct and simple scripts for data processing, system orchestration, and other automation tasks [21]. Consider for example the script shown in Figure 1, which is based on the original `spell` program by Johnson [1], lightly modified for modern environments. As this example illustrates, the Unix shell promotes a model of computation in which each command executes sequentially, with pipelined parallelism available between commands executing in the same pipeline. This pipelined computation model ignores parallelism that can be achieved by processing different parts of the input data in parallel, *i.e.*, data parallelism.

This fact is known in the Unix community and has motivated the development of a variety of tools that attempt to automatically exploit data parallelism in shell scripts [9, 11, 25, 28]. Unfortunately, these tools require manual effort and can easily generate parallel implementations that produce incorrect results. This is partially due to the fact that these tools do not accurately model shell semantics, but rather rely on operational intuition for correctness.

To address this challenge, I am working on PaSh, a new shell that focuses on exposing latent data parallelism in POSIX shell scripts. A main design goal of PaSh is to be provably correct, *i.e.*, achieve paralellism and improved performance without breaking the semantics of shell scripts. PaSh borrows ideas from prior research on dataflow models [6, 13, 14, 16, 19, 20] and automatic parallelization [7, 8, 23, 24], but also proposes its own set of solutions for the particular challenges present in the context of the shell.

2 CHALLENGES AND SOLUTIONS

2.1 Challenges

The shell comes with a unique set of challenges that make automatic parallelization difficult.

Subtle Command Data Parallelism Techniques used for achieving data parallelism in existing dataflow based systems, like MapReduce [5] and Apache Spark [29], require that dataflow nodes are commutative, i.e., the order in which they read their inputs does not affect the output, or data parallel with respect to some key, i.e., inputs with different keys can be processed independently. Such techniques are not directly applicable in the shell because the order in which commands read their inputs matters, for example `grep -vx -f $DICT` needs to complete reading `$DICT` before starting to read from its standard input.

Arbitrary Commands In contrast to restricted programming frameworks that enable parallelization by supporting a few carefully designed primitives, the Unix shell provides an unprecedented number and variety of composable commands—the example in Figure 1 composes 7 unique commands. These commands are written in a variety of programming languages, and the source is

Author's address: Konstantinos KallasUniversity of Pennsylvania, kallas@seas.upenn.edu.

```
cat $DIR/* | tr A-Z a-z | tr -d[:punct:] | sort | uniq | grep -vx -f $DICT > out ;
cat out | wc -l | sed 's/$/ mispelled words!/'
```

Fig. 1. **Spell:** The script streams two markdown files into a pipeline that converts characters in the stream into lower case, removes punctuation, sorts the stream in alphabetical order, removes duplicate words, and filters words based on whether they exist in a dictionary file. A second pipeline (line 4) counts the resulting lines to report the number of misspelled words to the user.

often not available, making an automated analysis that identifies parallelizability properties of commands infeasible. Furthermore, commands are continuously updated and users install new ones in their system, meaning that a one-time manual analysis of existing commands would quickly become obsolete.

Dynamic Environment The shell is a environment where the results of execution depend on several dynamic components, such as the file system, the current directory, environment variables, and unexpanded strings. The example in Figure 1 illustrates these dynamic features: the first `cat` reads an unknown number of files using `*` from an unknown directory `$DIR`, which could be relative to the current working directory `$PWD`. This makes it extremely hard to perform any safe analysis or transformation ahead of time.

2.2 Solutions

To address the three aforementioned challenges, we propose the following three solutions.

Order-aware dataflow model and Transformations We have developed an order-aware dataflow model that exposes information about the order in which nodes consume their inputs, such as the input consumption order of `grep -vx -f` in Figure 1, capturing a fragment of the shell that is pure, *i.e.*, scripts in this fragment only affect their environment through a set of output files. We build on this model, by developing several transformations that can be applied on dataflow programs and improve performance by exposing parallelism. We have proven that these transformations are correct, meaning that they preserve the behavior of the dataflow program, and we have formalized a bidirectional transformation from shell scripts in this fragment to our dataflow model and back.

Command-Node Correspondence Framework The expressiveness of the order-aware dataflow model enables defining a correspondence between shell commands and dataflow nodes, which is then lifted to a correspondence between dataflow programs and shell programs written in the pure fragment. We build on this by developing a framework that allows describing this correspondence for each command; the framework captures high-level properties of commands that are necessary for (i) translating a command to a dataflow node and back, e.g., its inputs and outputs and how they map to the command arguments, and (ii) the parallelizing transformations to be sound, e.g., whether the command processes each line of its input independently. We conducted a thorough study of all POSIX and GNU Coreutils commands to identify common command patterns, and based on those we designed an annotation language that succinctly captures the correspondence of several commands. The correspondence for each command, using the annotations or the general framework, can be created by command experts or automated tools, which can then be shared across shell users in the form of a command correspondence library. The correspondence library acts as an intermediate level abstraction that decouples the analysis of arbitrary shell commands from the development of tools that need to know whether each command satisfies a few high-level properties. To address cold-start issues, we wrote annotations for 47 commands (708 lines of annotation code).

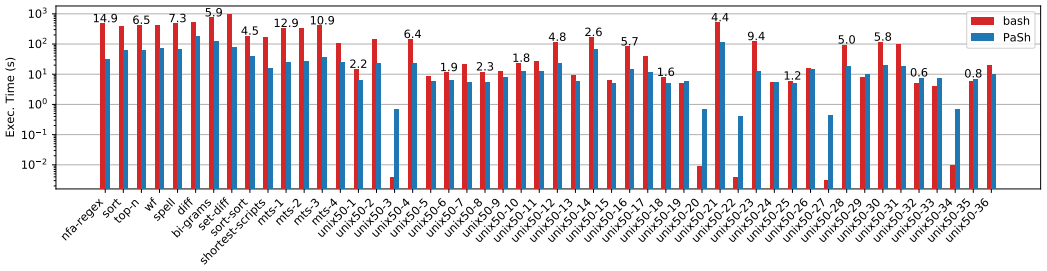

Fig. 2. **Execution time of PaSh and bash.** Average speedup of PaSh over bash is 4.15× (up to 14.93× maximum).

Just-in-Time Compilation To address the third challenge we have developed a just-in-time[1] compilation framework that is tightly coupled with the execution of the shell. Shell script execution is unique [10] in that it switches between (i) word expansion, e.g., expanding variables and *, and (ii) evaluation, e.g., executing a command by forking a worker process. Intuitively, our JiT compiler leaves all expansion, which is not computationally heavy but maintains a very complex state, to the user's shell, only interjecting before a switch from expansion to evaluation. At that point, given the current execution state, e.g., environment variables and the state of the file system, if it can infer that the part of the script that is to be evaluated is amenable to parallelization, it compiles it, just-in-time, and then executes the generated parallel version of it. This allows the compiler to exploit all available dynamic information to generate a parallel version of the script that is both correct and efficient.

3 IMPLEMENTATION AND EVALUATION

We compose all these solutions in a system that parallelizes shell scripts called PaSh and is publicly available here: https://github.com/andromeda/pash. PaSh executes a given script using JiT compilation, compiles parts of it to dataflow programs in our model, optimizes them by applying parallelization transformations, and finally translates them back to shell scripts and executes them. In addition to the above solutions, our implementations contains solutions to several technical challenges, such as the development of a set of highly optimized primitives for buffering and data splitting that significantly improve processor utilization and the parallel script execution time.

3.1 Evaluation

We briefly illustrate the execution time benefits achieved by PaSh, by using it to execute three sets of shell scripts taken from various sources, including GitHub, Stackoverflow, and the Unix literature [1–3, 12, 21, 26]:

- **Expert Scripts:** The first set contains 10 scripts: nfa-regex, sort, top-n, wf, spell, difference, bi-grams, set-difference, sort-sort, and shortest-scripts. These scripts contain commands that range from a scalable CPU-intensive `grep` in NFA-regex to a non-parallelizable `diff` in Difference. The input dataset of all these scripts (except for shortest-scripts) is the text of War and Peace downloaded by Project Gutenberg and multiplied to reach 1GB in size. The input data of shortest-scripts consists of all the commands in `/usr/bin` of the machine where the evaluation was executed multiplied 100 times to reach 8.5MB in size.
- **COVID-19 Mass-Transit Analysis Scripts:** The second set contains 4 scripts that were used to analyze real telemetry data from bus schedules during the COVID-19 response in one of Europe's

[1]Our definition of JiT compilation is literal and it does not involve identification of code hotspots or a warm up period.

largest cities [27]. The scripts compute several statistics on the transit system per day—such as average serving hours per day and average number of vehicles per day. The input dataset of the scripts is about 3.4GB and contains telemetry data from January 20th 2020 to January 9th 2021.

- **Unix50 Pipelines:** The third set contains 36 pipelines solving the Unix50 game [18]. This set is from a recent set of challenges celebrating of Unix's 50-year legacy, solvable by Unix pipelines. We found unofficial solutions to all-but-three problems on GitHub [3], expressed as pipelines with 2–12 stages (avg.: 5.58), and we executed them as-is without modifications. The input dataset of the scripts corresponds to the original Unix50 game input multiplied to reach 10GB in size.

Figure 2 shows the execution times achieved by PaSh and bash for all scripts on a 16-core machine. PaSh achieves significant speedups on all scripts that perform non-trivial computation (>0.1s).

4 BROADER IMPACT

Shell scripts are ubiquitous, utilize programs from a plethora of programming languages, and spend a significant fraction of their time executing sequentially. This paper present PaSh, a system that addresses several challenges of the domain of the shell to enable shell users to parallelize their programs mostly automatically.

Even though shell scripts are ubiquitous, they have mostly escaped the attention of the programming language community because of three main reasons: (i) shell commands can have arbitrary behaviors and are usually black boxes, making them hard to analyze or reason about, (ii) the shell's dynamic nature makes any static analysis or transformation incorrect or ineffective, and (iii) the shell's semantics and behavior is considered black magic and was until recently not fully understood. Recent work by Greenberg and Blatt [10] addressed the third issue, paving the way for a better understanding of the shell and the development of tools, analyses, and transformations targeting it. We see PaSh as a step towards addressing issues (i) and (ii) above, by (i) proposing a correspondence framework that enables decoupling command analysis from the develoment of tools for the shell, and by (ii) introducing a JiT compilation framework for the shell that allows performing analyses and transformations on shell scripts at the right time to have necessary runtime information. These two solutions create a foundation that can enable further studies of the performance and correctness of shell scripts, even outside of the domain of parallel computation. Two particularly interesting avenues for future work are:

Incremental Computation: IC has been studied for several domains [22] and is particularly useful in the context of the shell, since developing a script is an iterative cycle of coding, testing, and inspecting. In addition, scripts are often used for data-downloading, extracting, cleaning, etc. Combining the correspondence framework, which can expose necessary information for shell commands (such as their inputs and outputs), and the JiT compiler, which can have up-to-date information about the files in the file system, we have the critical building blocks for a runtime that incrementally reinterprets a script given changes in its input.

Developer Support: Shell script development is particularly hard due to its highly dynamic nature and the unpredicability of commands. Extending the correspondence framework with additional properties could allow for a variety of analyses—substantially more than syntactic checks [17], man-page-directed listings [15], and purely text-based analyses [4]. Such analyses could then be executed in an interactive way, by building on top of the JiT framework to have access to the latest relevant information.

REFERENCES

[1] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[2] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[3] Pawan Bhandari. Solutions to unixgame.io, 2020. Accessed: 2020-04-14.

[4] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 582âĂŞ592, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[7] Azadeh Farzan and Victor Nicolet. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 540–555, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 610–624, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[10] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the posix shell: Smoosh: the symbolic, mechanized, observable, operational shell. *Proc. ACM Program. Lang.*, 4(POPL):43:1–43:30, January 2020.

[11] Lluis Batlle i Rossell. *tsp(1) Linux User's Manual*. https://vicerveza.homeunix.net/ viric/soft/ts/, 2016.

[12] Dan Jurafsky. Unix for poets, 2017.

[13] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[14] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. *Information Processing*, 77:993–998, 1977.

[15] Idan Kamara. explainshell, 2016.

[16] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[17] koalaman. Shellcheck, 2016.

[18] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[19] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[20] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[21] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[22] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, 1993.

[23] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. Association for Computing Machinery.

[24] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 326–340, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Ole Tange. Gnu parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[26] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[27] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens. https://insidestory.gr/article/noymera-leoforeia-athinas?token=0MFVISB8N6, 2021.

[28] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.