PaSh: Light-touch Data-Parallel Shell Processing

Nikos Vasilakis[*]
MIT

Konstantinos Kallas[*]
University of Pennsylvania

Konstantinos Mamouras
Rice University

Achilleas Benetopoulos
Unaffiliated

Lazar Cvetkovich
Unaffiliated

Abstract

This paper presents PaSh, a system for parallelizing POSIX shell scripts. Given a script, PaSh converts it to a dataflow graph, performs a series of semantics-preserving program transformations that expose parallelism, and then converts the dataflow graph back into a script—one that adds POSIX constructs to explicitly guide parallelism coupled with PaSh-provided Unix-aware runtime primitives for addressing performance- and correctness-related issues. A lightweight annotation language allows command developers to express key parallelizability properties about their commands. An accompanying parallelizability study of POSIX and GNU commands—two large and commonly used groups—guides the annotation language and optimized aggregator library that PaSh uses. PaSh's extensive evaluation over 44 unmodified Unix scripts shows significant speedups (0.89–61.1×, avg: 6.7×) stemming from the combination of its program transformations and runtime primitives.

1 Introduction

The Unix shell is an environment—often interactive—for composing scripts written in a plethora of programming languages. This language-agnosticism, coupled with Unix's toolbox philosophy [30], makes the shell the primary choice for specifying succinct and simple scripts for data processing, system orchestration, and other automation tasks. Unfortunately, parallelizing such pipelines requires significant effort shared between two different programmer groups.

The first group is *command developers*, responsible for implementing individual commands such as `sort`, `uniq`, and `jq`. These developers usually work in a single programming language, leveraging its abstractions to provide parallelism whenever possible. As they have no visibility into the command's uses, they expose a plethora of ad-hoc command-specific flags such as `-t`, `--parallel`, `-p`, and `-j` [31, 38, 46].

The second group is *shell users*, who use POSIX shell constructs to combine multiple such commands from many languages into their scripts and are thus left with only a few options for incorporating parallelism. One option is to use manual tools such as GNU `parallel` [48], `ts` [20], `qsub` [13], SLURM [52]; these tools are either command-unaware, and thus at risk of breaking program semantics, or too coarse-grained, and thus only capable of exploiting parallelism at the level of entire scripts rather than individual components. Another option is to use shell primitives (such as `&`, `wait`) to explicitly induce parallelism; these come at a cost of manual effort to split inputs, rewrite scripts, and orchestrate execution—an expensive and error-prone process. To top it off, all these options assume a good understanding of parallelism; users with domain of expertise outside computing—from hobbyists to data analysts—are left without options.

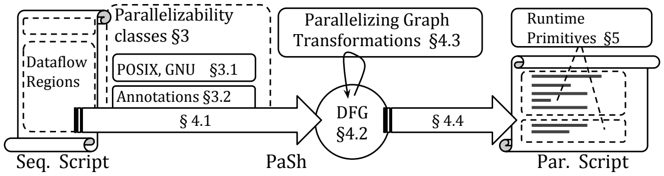


Fig. 1. PaSh overview. PaSh identifies dataflow regions (§4.1), converts them to dataflow graphs (§4.2), applies transformations (§4.3) based on the parallelizability properties of the commands in these regions(§3.1, §3.2), and emits a parallel script that uses custom primitives (§5).

To address this challenge, we develop a system called PaSh and outlined in Fig. 1 for parallelizing POSIX shell scripts. PaSh benefits both programmer groups, with emphasis on users. Command developers are given a set of abstractions, akin to lightweight type annotations, for expressing the parallelizability properties of their commands: rather than expressing a command's full observable behavior, these annotations focus primarily on its interaction with state. Shell users, on the other hand, are provided with full automation: PaSh analyzes their scripts and extracts latent parallelism. PaSh's transformations are conservative, in that they do not attempt to parallelize fragments that lack sufficient information—*i.e.*, at worst, PaSh will choose to not improve performance rather than risking breakage.

To address cold-start issues, PaSh comes with a library of parallelizability annotations for commands in POSIX and GNU Coreutils. These large classes of commands serve as the shell's standard library, expected to be used pervasively. The study that led to their characterization also informed PaSh's annotation and transformation components.

The above are tied together with PaSh's runtime component. Aware of the Unix philosophy and abstractions, it packs a small library of highly-optimized data aggregators as well as high-performance primitives such as ones for eager data splitting and merging. These address many practical challenges and were developed by uncovering several pathological situations, on a few of which we report.

---

[*] The two marked authors contributed equally to the paper.

We evaluate PaSh on 44 unmodified scripts including: (i) a series of benchmarks, ranging from classic Unix one-liners to modern data-processing scripts; (ii) two large and complex use cases for temperature analysis and web indexing; and (iii) two micro-benchmarks inspired by (and comparing with) possible alternatives. Speedups range between 0.89–61.1× (avg: 6.7×), with the 39 out of 44 scripts seeing non-trivial speedups. PaSh's runtime primitives add to the base speedup extracted by PaSh's program transformations—*e.g.*, 8.83× over a base 5.93× average for 10 representative Unix one-liners. PaSh accelerates a large program for temperature analysis by 2.52×, parallelizing both the computation (12.31×) and the preprocessing (2.04×) fragment (*i.e.*, data download, extraction, and cleanup), which traditionally falls outside of the focus of conventional parallelization systems—even though it takes 75% of the total execution time.

The paper is structured as follows. It starts by introducing the necessary background on shell scripting and overviewing PaSh (§2). Sections 3–5 highlight key contributions:

- §3 studies the parallelizability of shell commands, and introduces a lightweight annotation language for commands that are executable in a data-parallel manner.
- §4 presents a dataflow model and associated transformations that expose data parallelism while preserving the semantics of the sequential program.
- §5 details PaSh's runtime component, discussing the correctness and performance challenges it addresses.

After PaSh's evaluation (§6) and comparison with related work (§7), the paper concludes (§8).

2 Background and Overview

This section reviews Unix shell scripting through an example (§2.1), which it then uses to present parallelization challenges (§2.2) and how they are addressed by PaSh (§2.3).

2.1 Running Example: Weather Analysis

Suppose an environmental scientist wants to get a quick sense of trends in the maximum temperature across the U.S. over the past five years. As the National Oceanic and Atmospheric Administration (NOAA) has made historic temperature data publicly available [35], answering this question is only a matter of a simple data-processing pipeline.

Fig. 2's script starts by pulling the yearly index files and filtering out URLs that are not part of the compressed dataset. It then downloads and decompresses each file in the remaining set, extracts the values that indicate the temperature, and filters out bogus inputs marked as `999`. It then calculates the maximum yearly temperature by sorting the values and picking the top element. Finally, it matches each maximum value with the appropriate year in order to print the result. The effort expended writing this script is low: its data-processing core amounts to 12 stages and, when expressed as a single line, is only 165 characters long. This program is no toy: a Java program implementing only the last four stages takes 137 LoC [51, §2.1]. To enable such a succinct program composition, Unix incorporates several features.

```
base="ftp://ftp.ncdc.noaa.gov/pub/data/noaa";
for y in {2015..2019}; do
 curl $base/$y | grep gz | tr -s" " | cut -d" " -f9 |
 sed "s;^;$base/$y/;" | xargs -n 1 curl -s | gunzip |
 cut -c 89-92 | grep -iv 999 | sort -rn | head -n 1 |
 sed "s/^/Maximum temperature for $y is: /"
done
```

Fig. 2. Calculating maximum temperatures per year. The script downloads daily temperatures recorded across the U.S. for the years 2015–2019 and extracts the maximum for every year.

Unix Features Composition in Unix is primarily achieved with pipes (`|`), a construct that allows for task-parallel execution of two commands by connecting them through a character stream. This stream is comprised of contiguous character lines separated by newline characters (NL) delineating individual stream elements. For example, Fig 2's first `grep` outputs (file-name) elements containing `gz`, which are then consumed by `tr`. A special end-of-file (EOF) condition marks the end of a stream.

Different pipeline stages process data concurrently and possibly at different rates—*e.g.*, the second `curl` produces output at a significantly slower pace than the `grep` commands before and after it. The Unix kernel facilitates scheduling, communication, and synchronization behind the scenes.

Command flags, used pervasively in Unix, are configuration options that the command's developer has decided to expose to its users to improve the command's general applicability. For example, by omitting `sort`'s `-r` flag that enables reverse sorting, the user can easily get the minimum temperature. The shell does not have any visibility into these flags; after it expands special characters such as `~` and `*`, it leaves parsing and evaluation entirely up to individual commands.

Finally, Unix provides an environment for composing commands written in any language. Many of these commands come with the system—*e.g.*, ones defined by the POSIX standard or ones part of the GNU Coreutils—whereas others are available as add-ons. The fact that commands are developed in a variety of languages—including shell scripts—provides users with significant flexibility. For example, one could replace `sort` and `head` with `./avg.py` to get the average rather than the maximum—the pipeline still works, as long as `./avg.py` conforms to the interface outlined earlier.

2.2 Parallelization Challenges

While these features aid development-effort economy through powerful program composition, they complicate shell script parallelization, which even for simple scripts such as the one in Fig. 2 create several challenges.

Commands In contrast to restricted programming frameworks that enable parallelization by supporting a few carefully-designed primitives [6, 9, 14, 53], the Unix shell provides an unprecedented number and variety of composable commands. To be parallelized, each command may require special analysis and treatment—*e.g.*, exposing data parallelism in Fig. 2's `tr` or `sort` would require splitting their inputs, running them on each partial input, and then merging the partial results.[1] Automating such an analysis is infeasible, as individual commands are black boxes written in a variety of programming languages and models. Manual analysis is also challenging, due to the sheer number of commands and the many flags that affect their behavior—*e.g.*, Fig. 2's `cut` is called with two flag sets.

Scripts Another challenge is due to the language of the POSIX shell. First, it contains constructs that enforce sequential execution: Fig.2's sequential composition operator (`;`) means that the assignment to `base` is completed before everything else. Moreover, the language semantics only exposes limited task-based parallelism in the form of constructs such as `&`. Even though Fig. 2's **`for`** focuses only on five years of data, `curl` still outputs thousands of lines per year; naive parallelization of each loop iteration will miss such opportunities. Any attempt to automate parallelization should be aware of the POSIX shell language, exposing latent data parallelism without modifying execution semantics.

Implementation On top of command and shell semantics, the broader Unix environment has its own set of quirks. Any attempt to orchestrate parallel execution will hit challenges related to task parallelism, deadlock prevention, and runtime performance. For example, forked processes piping their combined results to Fig. 2's `head` may not receive a `PIPE` signal if `head` exits prior to opening all pipes. Moreover, several commands such as `sort` and `uniq` require specialized data aggregators in order to be correctly parallelized.

2.3 PaSh Design Overview

At a high level, PaSh takes as input a POSIX shell script like the one in Fig. 2 and a parameter `--width`, and outputs a new script, a fragment of which is shown in Fig. 3—for `--width=2`, *i.e.*, 2× {`grep`, `tr`, `cut`, *etc.* }. PaSh first identifies sections of the script that lack synchronization and scheduling constraints and converts them to dataflow graphs (DFGs). It then performs a series of DFG transformations that expose parallelism without breaking semantics, by expanding the DFG to the desired `width`. Finally, PaSh converts these DFGs back to a shell script augmented with PaSh-provided commands. The script is handed off to the user's original shell interpreter for execution. PaSh addresses the aforementioned challenges (§2.2) as below.

```
mkfifo $t{0,1...}
curl $base/$y > $t0 & cat $t0 | split $t1 $t2 &
cat $t1 | grep gz > $t3 &
cat $t2 | grep gz > $t4 &
...
cat $t9 | sort -rn > $t11 & cat $t10 | sort -rn > $t12 &
cat $t11 | eager > $t13 & cat $t12 | eager > $t14 &
sort -mrn $t13 $t14 > $t15 &
cat $t15 | head -n1 > $out1 &
wait $! && get-pids | xargs -n 1 kill -SIGPIPE
```

Fig. 3. Output of `pash --width=2` for Fig. 2 (fragment). PaSh orchestrates parallel execution through named pipes, parallel operators, and custom runtime primitives—*e.g.*, `eager`, `split`, and `get-pids`.

Commands To understand standard commands available in any shell, PaSh groups POSIX and GNU commands into a small but well-defined set of *parallelizability classes* (§3.1). Rather than describing a command's full observable behavior, these classes focus on information that is important for data parallelism. To allow other commands to use its transformations, PaSh defines a light *annotation language* for describing a command's parallelizability class (§3.2). Annotations are expressed once per command rather than once per script and aim towards command developers rather than its users, so that they can quickly and easily capture the characteristics of the commands they develop.

Scripts To maintain sequential semantics, PaSh first analyzes a script to identify *dataflow regions* containing commands that are candidates for parallelization (§4.1). This analysis is guided by the script structure: some constructs expose parallelism; others enforce synchronization. PaSh then converts each dataflow region to a *dataflow graph* (DFG) (§4.2), a flexible representation that enables a series of local transformations to expose data parallelism, converting the graph into its parallel equivalent (§4.3). Further transformations compile the DFG back to a shell script that uses POSIX constructs to guide parallelism explicitly while aiming at preserving the semantics of the sequential program (§4.4).

Implementation PaSh addresses several practical challenges through a set of constructs it provides—*i.e.*, modular components for augmenting command composition (§5). It also provides a small and efficient *aggregator library* targeting a large set of parallelizable commands. All these commands live in the `PATH` and are addressable by name, which means they can be used like (and by) any other commands.

3 Parallelizability Classes

PaSh aims at parallelizing data-parallel commands, *i.e.*, ones that can process their input in parallel, encoding their characteristics by assigning them to *parallelizability classes*. PaSh leans towards having a few coarse classes rather than many detailed ones—among other reasons, to simplify their understanding and use by command developers.

This section starts by defining these classes, along with a parallelizability study of the commands in POSIX and GNU

---

[1] As explained in the introduction (§1), commands such as `sort` may have *ad hoc* flags like `--parallel`, which do not compose across commands and may risk breaking correctness and unexploiting performance potential (§6.5).

Tab. 1. Parallelizability Classes. Broadly, Unix commands can be broken down into to four classes according to their parallelizability properties.

Class	Key	Examples	Coreutils	POSIX
Stateless	Ⓢ	`tr`, `cat`, `grep`	22 (21.1%)	28 (18%)
Parallelizable Pure	Ⓟ	`sort`, `wc`, `uniq`	8 (7.6%)	9 (5%)
Non-parallelizable Pure	Ⓝ	`sha1sum`	13 (12.4%)	13 (8.3%)
Side-effectful	Ⓔ	`env`, `cp`, `whoami`	57 (58.8%)	105(67.8%)

Coreutils (§3.1). Building on this study, it develops a lightweight annotation language that enables command classification by its developers (§3.2). PaSh in turn uses this language to annotate POSIX and GNU commands and generate their wrappers, as presented in later sections.

3.1 Parallelizability of Standard Libraries

Broadly speaking, shell commands can be split into four major classes with respect to their parallelization characteristics, depending on what kind of state they mutate when processing their input (Tab.1). These classes are ordered in ascending difficulty (or impossibility) of parallelization. In this order, some classes can be thought as subsets of the next—*e.g.*, all stateless commands are pure—meaning that the synchronization mechanisms required for any superclass would work with its subclass (but foregoing any performance improvements). Commands can change classes depending on their flags, which are discussed later (§3.2).

Stateless Commands The first class, Ⓢ, contains commands that operate on individual line elements of their input, without maintaining state across invocations. These are commands that can be expressed as a purely functional *map* or *filter*—*e.g.*, `grep` filters out individual lines and `basename` removes a path prefix from a string. They may produce multiple elements—*e.g.*, `tr` may insert NL tokens—but always return empty output for empty input. Workloads that use only stateless commands are trivial to parallelize: they do not require any synchronization to maintain correctness, nor caution about where to split inputs.

The choice of line as the data element strikes a convenient balance between coarse-grained (files) and fine-grained (characters) separation while staying in line with Unix's core abstractions. This choice can affect the allocation of commands in Ⓢ, as many of its commands (about 1/3) are stateless *within* a stream element—*e.g.*, `tr` transliterates characters within a line, one at a time—enabling further parallelization by splitting individual lines. This feature may seem of limited use, as these commands are computationally inexpensive, precisely due to their narrow focus. However, it may be useful for cases with very large stream elements (*i.e.*, long lines) such the `.fastq` format used in bioinformatics.

Parallelizable Pure Commands The second class, Ⓟ, contains commands that respect functional purity—*i.e.*, same outputs for same inputs—but maintain internal state across their entire pass. The details of this state and its propagation during element processing affect their parallelizability characteristics. Some commands are easy to parallelize, because they maintain trivial state and are commutative—*e.g.*, `wc` simply maintains a counter. Other commands, such as `sort`, maintain more complex invariants that have to be taken into account when merging partial results.

Often these commands do not operate in an online fashion, but need to block until the end of a stream. A typical example of this is `sort`, which cannot start emitting results before the last input element has been consumed. Such constraints affect task parallelism, but not data parallelism: `sort` can be parallelized significantly using divide-and-conquer techniques—*i.e.*, by encoding it as a group of (parallel) *map* functions followed by an *aggregate* that merges the results.

Non-parallelizable Pure Commands The third class, Ⓝ, contains commands that, while purely functional, cannot be parallelized. This is because their internal state depends on prior state in the same pass in non-trivial ways. For example, hashing commands such as `sha1sum` maintain complex state that has to be updated sequentially. If parallelized on a single input, each stage would need to wait on the results of all previous stages, foregoing any parallelism benefits.

It is worth noting that while these commands are not parallelizable at the granularity of a single input, they are still parallelizable across different inputs. For example, a web crawler involving hashing to compare individual pages would allow `sha1sum` to proceed in parallel for different pages.

Side-effectful Commands The last class, Ⓔ, contains commands that have side-effects across the system—for example, updating environment variables, interacting with the filesystem, and accessing the network. Such commands are not parallelizable without finer-grained concurrency control mechanisms that can detect side-effects across the system.

This is the largest class, for two reasons. First, it includes commands related to the file-system—a central abstraction of the Unix design and philosophy [42]. In fact, Unix uses the file-system as a proxy to several file-unrelated operations such as access control and device driving. Second, it contains commands that do not consume input or do not produce output—and thus are not amenable to data parallelism. For example, `date`, `uname`, and `finger` are all commands interfacing with kernel- or hardware-generated information and do not consume any input from user programs.

3.2 Extensibility Annotations

To address the challenge of language-agnostic extensibility (§2), PaSh allows communicating several key details about command parallelizability through lightweight annotations. These annotations can be used by both developers of new commands—including users developing their own scripts—as well as developers maintaining existing commands. The latter can express additions or changes to the command's implementation or interface, important as commands are maintained or extended over long periods of time.

Key Concerns PaSh's annotations focus on three crucial concerns about a command: (C1) its parallelizability class, (C2) its inputs and outputs, and the characteristics of its input consumption, and (C3) how flags affect its class, inputs, and outputs. The first concern was discussed extensively in the previous section; we now turn to the latter two.

To be able to accurately construct the DFG representation, PaSh needs to know certain details about a command's inputs and outputs. There are a few reasons for this. One reason is due to PaSh's DFG representation, which connects commands with each other. To perform this connection correctly, PaSh needs to know the outputs of a command. A second reason is due to ordering: as some commands consume their inputs in a certain order, this order needs to be maintained in the parallel version of the program. For example, consider the data-parallel version of `grep "foo" f1 f2`:

```
mkfifo t1 t2
grep "foo" f1 > t1 & grep "foo" f2 > t2 & cat t1 t2
```

This is correct only because PaSh knows that `grep` in the sequential program reads first from `f1` and then from `f2`.

Command flags (options) are a particularly popular way of controlling a command's execution, directly affecting its parallelizability classification. Commands are thus assigned a default parallelizability class, which is then refined by the set of flags the command uses. For example, `cat` defaults to Ⓢ, but with `-n` it jumps into Ⓟ because it has to keep track of a counter and print it along with each line.

Example Annotations The parallelizability properties of each command are specified in an annotation record that contains several components, each addressing one of the key concerns C1–3. At the top level, each annotation record contains a set of clauses, each one identified by a predicate on the command arguments (C3). Each clause contains an assignment for the command, indicating its class (C1), and the sequence of its inputs and outputs (C2).

For example, consider `comm`, which performs a join-like operation on its two inputs to identify common elements. When `comm` is invoked without any flags, it produces a three-column output: the first column contains lines that are unique to the first input, the second column contains lines that are unique to the second input, and the last column contains lines that exist in both inputs. A user can invoke `comm` with any combination of flags `-1`, `-2`, or `-3` to suppress the corresponding output column(s).

Fig. **??** presents its annotation record (left) and the implementation of the record's first clause (right).

```
comm {
 | -1 ∧ -3 ⇒ (S, [arg[1]], [stdout])
 | -2 ∧ -3 ⇒ (S, [arg[0]], [stdout])
 | _       ⇒
     (P, [arg[0], arg[1]], [stdout])
}
```

```
"predicate":
  {
    "operator": "all",
    "operands": ["-1", "-3"]
  },
"class": "stateless",
"inputs": ["args[1]"],
"outputs": ["stdout"],
"options": ["stdin-hyphen",
            "short-combined"]
```

Fig. 4. Parallelizability annotation. Annotation record for `comm` (left) and the implementation of the record's first clause (right).

As the first and third columns are suppressed, `comm` can be categorized in Ⓢ by regarding its first file argument as a static "configuration" input; this input accompanies every replicated instance of the command executing in parallel. The second clause is symmetric to the first with respect to the second input. The third clause is the general case, classifying `comm` in Ⓟ, as `comm` needs to read both inputs completely before outputting its results. Option `stdin-hyphen` indicates that `-` represents standard input if given as a file argument, and option `short-combined` indicates that flags can be combined, as in `comm -13`.

Complete Annotation Language The complete annotation language contains 6 operators, one of which supports regular expressions. It was used to annotate 47 commands, totaling 708 lines of JSON—an effort that took about 2–3 hours. Annotation records are by default conservative so as to not jeopardize correctness, but can be incrementally refined to capture parallelizability when using increasingly complex combinations of flags. The language is extensible with more operators (as long as the developer defines their semantics); it also supports writing arbitrary Python code for commands whose properties are difficult to capture—*e.g.*, higher order `xargs`, whose parallelizability class depends on the class of the first-order command that it invokes.

Custom Aggregators For commands in Ⓢ, the annotations are enough to enable parallelization: commands are applied to parts of their input in parallel, and their outputs are simply concatenated. To support the parallelization of arbitrary commands in Ⓟ, PaSh allows supplying custom *map* and *aggregate* functions. In line with the Unix philosophy, these functions can be written in any language as long as they conform to a few invariants: (i) *map* is in Ⓢ and *aggregate* is in Ⓟ, (ii) *map* can consume (or extend) the output of the original command and *aggregate* can consume (and combine) the results of multiple *map* invocations, and (iii) their composition produces the same output as the original command. PaSh can use the *map* and *aggregate* functions in its graph transformations (§4) to further expose parallelism. PaSh defines aggregators for many POSIX and GNU commands in Ⓟ, doubling as both its standard library and an exemplar for community efforts tackling other commands.

4 Dataflow Graph Model

PaSh's core is an abstract dataflow graph (DFG) model (§4.2) that is used as the intermediate representation on which PaSh performs parallelization-exposing transformations (§4.3). PaSh first lifts sections of the input script to the DFG representation (§4.1), then performs transformations to expose

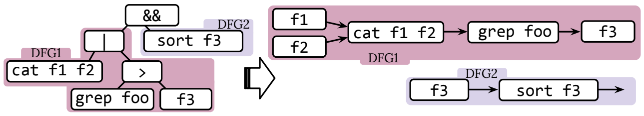


Fig. 5. From a script AST to DFGs. The AST on the left has two dataflow regions, each not extending beyond && (*Cf.*§4.1).

parallelism (up to the desired `--width`) (§4.3), and finally instantiates each DFG back to a parallel shell script (§4.4). A fundamental difference with other DFG models is that PaSh's DFG encodes the order in which a node reads its input streams (not just the order of input elements per stream), which in turn enables a set of graph transformations that can be iteratively applied to expose parallelization opportunities for Ⓢ and Ⓟ commands.

4.1 Frontend: From a Sequential Script to DFGs

Dataflow Regions In order to apply the graph transformations that expose data parallelism, PaSh first has to identify program sub-expressions that can be safely transformed to a dataflow graph, *i.e.*, sub-expressions that (i) do not impose any scheduling or synchronization constraints (*e.g.*, by using `;`), and (ii) take a set of files as inputs and produce a set of files as outputs. The search for these regions is guided by the shell language and the structure of a particular program. These contain information about (i) fragments that can be executed independently, and (ii) barriers that are natural synchronization points. Consider this code fragment:

```
cat f1 f2 | grep "foo" > f3 && sort f3
```

The `cat` and `grep` commands execute independently (and concurrently) in the standard shell, but `sort` waits for their completion prior to start (Fig. 5). Intuitively, dataflow regions correspond to sub-expressions of the program that would be allowed to execute independently by different processes in the POSIX standard [17]. Larger dataflow regions can be composed from smaller ones using the pipe operator (`|`) and the parallel-composition operator (`&`). Conversely, all other operators, including sequential composition (`;`) and logical operators (`&&`, `||`), represent barrier constructs that do not allow dataflow regions to expand beyond them.

Translation Pass PaSh's front-end performs a depth-first search on the AST of the given shell program. During this pass, it extends the dataflow regions bottom-up, translating their independent components to DFG nodes until a barrier construct is reached. All AST subtrees not translatable to DFGs are kept as they are. The output of the translation pass is the original AST where dataflow regions have been replaced with DFGs.

To identify opportunities for parallelization, the translation pass extracts each command's parallelizability class together with its inputs and outputs. To achieve this for each command, it searches all its available annotations (§3.2) and resorts to conservative defaults if none is found. If the command is in Ⓢ, Ⓟ, or Ⓝ, the translation pass initiates a dataflow region that is propagated up the tree.

Due to the highly dynamic nature of the shell, some information is not known to PaSh at the time of translation. Examples of such information include the values of environment variables, strings that have not been expanded, and sub-shell constructs. For safety purposes, PaSh takes a conservative approach, avoiding parallelization of such nodes with incomplete information. It will not attempt to parallelize sub-expressions in which the translation pass cannot infer that, *e.g.*, an environment variable passed as an argument to a command does not change its parallelizability class.

4.2 Dataflow Model Definitions

The two main shell abstractions are (i) streams, *i.e.*, files or pipes, that contain data, and (ii) commands, communicating through streams. PaSh's DFG model represents commands as nodes and streams as edges. We now introduce the notation used in throughout this section. For a set D, we write D^* to denote the set of all finite words over D. For words $x, y \in D^*$, we write $x \cdot y$ or xy to denote their concatenation. We write ε for the empty word. We say that x is a *prefix* of y, and we write $x \leq y$, if there is a word z such that $y = xz$. The $\leq$ order is reflexive, antisymmetric, and transitive (*i.e.*, it is a partial order), and is often called the *prefix order*.

Edges—Streams Edges in the DFG represent streams, the basic data abstraction of the shell. They are used as communication channels between nodes in the graph, and as the input or output of the entire graph. Edges are represented as possibly unbounded streams of type D^*. Edges can either refer to named files or FIFO pipes used for interprocess communication. Edges that do not start from a node in the graph represent the graph input; edges that do not point to a node in the graph represent its outputs.

Nodes—Commands A node f of the graph represents a function from a (possibly empty) list of inputs to a list of outputs $f : [D^*] \rightarrow [D^*]$, where D represents the basic data type of a line of characters. This representation captures all the commands in the Ⓢ, Ⓟ, and Ⓝ. Note that these functions should only produce output in the form of files and not perform any other side effect, such as sending signals. We require that the function f is monotone with respect to a lifting of the prefix order for a sequence of inputs. This captures the idea that a node cannot retract output that it has already produced.

Streaming Commands A large subset of the parallelizable Ⓢ and Ⓟ classes falls into the special category of streaming commands. These commands consume their inputs sequentially and one element at a time, with the possible exception of static input files, and produce one output file. The quintessential example of streaming commands is `cat`, which consumes its inputs in order, producing their concatenation as output. A more interesting example is `comm -23 f1 f2`,

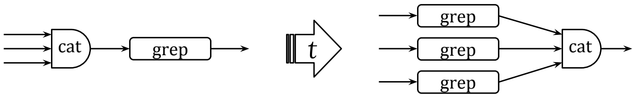


Fig. 6. Stateless parallelization transformation. The `cat` node is commuted with the stateless node to utilize available data parallelism.

which has `f2` as its static input (which it needs to read completely before consuming anything from `f1`, which is in turn consumed one element at a time). These commands can be represented as functions of type $f : D^* \times [D^*] \to D^*$, where the first argument represents the concatenation of the inputs that are consumed one at a time and the second argument represents the static inputs.

4.3 Graph Transformations

PaSh defines a set of semantics-preserving graph transformations that act as parallelization-exposing optimizations. Both the domain and range of these transformations is a graph in PaSh's DFG model and therefore transformations can be composed arbitrarily and in any order. Before describing the different types of transformations, we formalize the intuition behind classes Ⓢ and Ⓟ described informally earlier (§3.1).

Stateless and Parallelizable Pure Commands Stateless commands such as `tr` operate independently on individual elements of the stream (characters, lines, or files) without maintaining any state (§3.1). In this work, we consider lines as the data quantum, thus we consider stateless only commands that are stateless with respect to lines (or any finer granularity such as characters). Formally, a streaming command f is stateless if it commutes with the operation of concatenation, *i.e.*, it is a semigroup homomorphism:

$$\forall x, x', s, f(x \cdot x', s) = f(x, s) \cdot f(x', s)$$

This means that applying the command f to a concatenation of two inputs x, x' produces the same output as applying f to each input x, x' separately, and concatenating the outputs.

Similarly, some pure commands (such as `sort` and `wc`) can be parallelized using divide-and-conquer techniques. More formally, these pure commands f can be implemented as a combination of a function map m and an associative aggregate agg, satisfying the following equation:

$$\forall x, x', s, f(x \cdot x', s) = agg(m(x, s), m(x', s), s)$$

This means that we get the same output by applying f to a concatenation of two inputs x, x' as by applying the aggregation function agg to the outputs produced by applying map m to each of x and x'.

Parallelization Transformations Based on these equations, we can define a node parallelization transformation T on a node $v \in$ Ⓢ that is preceded by a concatenation, *i.e.*, the command `cat`, of n input streams and is followed by a node v' (Fig. 6). T replaces v with n new nodes, routing each of the n input streams to one of them, and commutes the `cat` node after them to concatenate their outputs and transfer them to v'. Since each incoming edge represents a stream of data $x_i : D^*$, and the only behavior of a DFG is its output, this optimization $v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow v(x_1, s) \cdot v(x_2, s) \cdots v(x_n, s)$ can be shown to preserve the behavior of the graph.

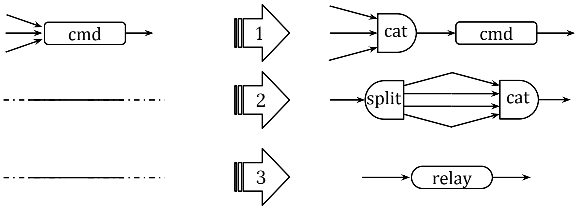


Fig. 7. Auxiliary transformations. These augment the DFG with `cat`, `split`, and `relay` nodes.

T can be extended straightforwardly to nodes $v \in$ Ⓟ, implemented by a map-aggregate pair (m, agg) as $v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow agg(m(x_1, s), m(x_2, s), \ldots m(x_n, s), s)$. As long as the pair (m, agg) meets the three invariants outlined earlier (§3.2), T can be shown to be behavior-preserving.

Auxiliary Transformations PaSh also performs a set of auxiliary transformations t_{1-3} that are depicted in Fig. 7. If a node has many inputs, t_1 concatenates these inputs by inserting a `cat` node to enable the parallelization transformations. In cases where a parallelizable node has one input and is not preceded by a concatenation, t_2 inserts a `cat` node that is preceded by its inverse `split`, so that the concatenation can be commuted with the node. Transformation t_3 inserts a relay node that performs the identity transformation. Relay nodes can be useful for monitoring and debugging, as well as for performance improvements (§5). The additional relay nodes inserted by the auxiliary transformations affect the resulting parallelism.

4.4 Backend: From DFGs to a Parallel Shell Script

After applying transformations (§4.3), PaSh translates all DFGs back into a shell script. Nodes of the graph are instantiated with the commands and flags they represent, and edges are instantiated as named pipes. A prologue in the script creates the necessary intermediate pipes, and a `trap` call takes care of cleaning up when the script aborts.

5 Runtime

This section describes technical challenges related to execution of the resulting script and how they are addressed by PaSh's custom runtime primitives.

Overcoming Laziness The shell's evaluation strategy is unusually lazy, in that most commands and shell constructs consume their inputs only when they are ready to process more. Such laziness leads to CPU underutilization, as commands are often blocked when their consumers are not requesting any input. Consider the following fragment:

```
mkfifo t1 t2
grep "foo" f1 > t1 & grep "foo" f2 > t2 & cat t1 t2
```

(a) No Eager —　(c) Blocking Eager —

(b) Wrong Eager ✘　(d) PaSh Eager ✓

Fig. 8. Eager primitive. Addressing intermediary laziness is challenging: (a) FIFOs are blocking; (b) files alone introduce race conditions between producer/consumer; (c) files + `wait` inhibit task parallelism. Eager `relay` nodes (d) address the challenge while remaining within the PaSh model.

The `cat` command will consume input from `t2` only after it completes reading from `t1`. As a result, the second `grep` will remain blocked until the first `grep` completes (Fig. 8a).

To solve this, one might be tempted to replace FIFOs with files, a central Unix abstraction, simulating pipes of arbitrary buffering (Fig. 8b). Aside from severe performance implications, naive replacement can lead to subtle race conditions as a consumer might reach EOF before a producer. Alternatively, consumers could wait for producers to complete before opening the file for reading (Fig. 8c); however, this would insert artificial barriers impeding task-based parallelism and wasting disk resources—that is, this allows for data parallelism at the expense of task parallelism.

To address this challenge, PaSh inserts and instantiates eager `relay` nodes at these points (Fig. 8d). These nodes feature tight multi-threaded loops that consume input eagerly while attempting to push data to the output stream, forcing upstream nodes to produce output when possible while also preserving task-based parallelism. In PaSh's evaluation (§6), these primitives have the names presented in Fig. 8.

Splitting Challenges PaSh's optimizer inserts `split` nodes to expose parallelism when parallelizable nodes only have one input (§4.3). For `split` to be effective, it needs to disperse its input uniformly across its outputs. This requires the input size to be known beforehand, which is not always the case. To address this, PaSh provides a `split` implementations that first consumes its complete input, counts its lines, and then splits it uniformly across the desired number of outputs. PaSh also inserts eager `relay` nodes after all `split` outputs (except the last one) to address laziness concerns.

Dangling FIFOs and Zombie Producers Under normal operation, a command exits after it has produced and sent all its results to its output channel. If the channel is a pipe and its reader exits early, the command is notified to stop writing early. In Unix, this is achieved by an out-of-band error mechanism: the operating system delivers a `PIPE` signal to the producer, notifying it that the pipe's consumer has exited. This is different from errors for other system calls and unusual compared to non-Unix systems[2] primarily because pipes and pipelines are at the heart of Unix. Unfortunately though, if a pipe has not been opened for writing yet, Unix cannot signal this condition. Consider the following script:

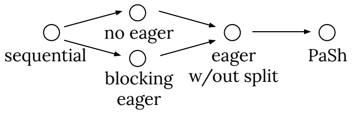


Fig. 9. Runtime setup lattice. Parallel No Eager and Blocking Eager improve over sequential, but are not directly comparable. Eager adds PaSh's optimized `eager` relay, and PaSh uses all primitives in §5 (Fig. 10).

```
mkfifo fifo1 fifo2
cat in1 > fifo1 & cat in2 > fifo2 &
cat fifo1 fifo2 | head -n 1 & wait
```

In the code above, `head` exits early causing the last `cat` to exit before opening `fifo2`. As a result, the second `cat` never receives a `PIPE` signal that its consumer exited—after all, `fifo2` never even had a consumer. This, in turn, leaves it unable to make progress, as it is both blocked and unaware of its consumer exiting. Coupled with `wait` at the end, the entire snippet reaches a deadlock.

To solve this problem, PaSh emits cleanup logic that operates from the end of the pipeline and towards its start. The emitted code first gathers the IDs of the output processes and passes them as parameters to `wait`; this causes `wait` to block only on the output producers of the dataflow graph. Right after `wait`, PaSh inserts a routine that delivers `PIPE` signals to any remaining processes upstream.

Aggregator Implementations Commands in Ⓟ can be parallelized using a *map* and an *aggregate* stage (§3). PaSh implements *aggregate* for several commands in Ⓟ to enable parallelization. A few interesting examples are *aggregate* functions for (i) `sort`, which amounts to the merge phase of a merge-sort (and on GNU systems is implemented as `sort -m`), (ii) `uniq` and `uniq -c`, which need to check conditions at the boundary of their input streams, (iii) `tac`, which consumes stream descriptors in reverse order, and (iv) `wc`, which adds inputs with an arbitrary number of elements (*e.g.*, `wc -lw` or `wc -lwc` *etc.*). The *aggregate* functions iterate over the provided stream descriptors, *i.e.*, they work with more than two inputs, and apply pure functions at the boundaries of input streams (with the exception of `sort` that has to interleave inputs).

6 Evaluation

This section reports on whether PaSh can indeed offer performance benefits automatically and correctly using several scripts collected out from the wild along with a few microbenchmarks for targeted comparisons.

Highlights This paragraph highlights results for `width=16`, but PaSh's evaluation reports on varying `width`s (2–64). Overall, applying PaSh to all 44 unmodified scripts accelerates 39 of them by 1.92–17.42×; for the rest, the parallel performance is comparable to the sequential (0.89, 0.91, 0.94, 0.99, 1.01×). The total average speedup over all 44 benchmarks is

---

[2] For example, Windows indicates errors for `WriteFile` using its return code—similar to `DeleteFile` and other Win32 functions.

Tab. 2. Summary of shell one-liners. Structure summarizes the different classes of commands used in the script. Input and seq. time report on the input size fed to the script and the timing of its sequential execution. Nodes and compile time report on PaSh's resulting DFG size (which is equal to the number of resulting processes and includes aggregators, `eager`, and `split` nodes) and compilation time for two indicative `--width`s.

Script	Structure	Input	Seq. Time	#Nodes(16, 64)		Compile Time (16, 64)		Highlights
nfa-regex	3 × Ⓢ	1 GB	79m35.197s	49	193	0.056s	0.523s	complex NFA regex
sort	Ⓢ, Ⓟ	10 GB	21m46.807s	77	317	0.090s	1.083s	`sorting`
top-n	2 × Ⓢ, 4 × Ⓟ	10 GB	78m45.872s	96	384	0.145s	1.790s	double `sort`, `uniq` reduction
wf	3 × Ⓢ, 3 × Ⓟ	10 GB	22m30.048s	96	384	0.147s	1.809s	double `sort`, `uniq` reduction
spell	4 × Ⓢ, 3 × Ⓟ	3 GB	25m7.560s	193	769	0.104s	1.038s	comparisons (`comm`)
shortest-scripts	5 × Ⓢ, 2 × Ⓟ	85 MB	28m45.900s	142	574	0.328s	4.657s	long Ⓢ pipeline ending with Ⓟ
difference	2 × Ⓢ, 3 × Ⓟ	10 GB	25m49.097s	125	509	0.186s	2.341s	non-parallelizable `diffing`
bi-grams	3 × Ⓢ, 3 × Ⓟ	3 GB	38m9.922s	185	761	0.146s	1.716s	stream shifting and merging
set-difference	5 × Ⓢ, 2 × Ⓟ	10 GB	51m32.313s	155	635	0.321s	4.358s	two pipelines merging to a `comm`
sort-sort	Ⓢ, 2 × Ⓟ	10 GB	31m26.147s	154	634	0.092s	1.077s	parallelizable Ⓟ after Ⓟ

6.7×. PaSh's runtime primitives offer significant benefits—for the 10 scripts that we measured with and without the runtime primitives they bump the average speedup from 5.9× to 8.6×. PaSh significantly outperforms `sort --parallel`, a hand-tuned parallel implementation, and performs better than GNU `parallel`, which returns incorrect results if used without care (§6.5).

Using PaSh's standard library of annotations for POSIX and GNU commands (§3), the vast majority of programs (> 40, with > 200 commands) require no effort to parallelize other than invoking PaSh; only 6 (< 3%) commands, outside this library, needed a single-record annotation (§6.4).

In terms of correctness, PaSh's results on multi-GB inputs are identical to the sequential ones. Scripts feature ample opportunities for breaking semantics (§6.5), which PaSh avoids.

Setup PaSh was run on 512GB of memory and 64 physical × 2.1GHz Intel Xeon E5-2683 cores, Debian 4.9.144-3.1, GNU Coreutils 8.30-3, GNU Bash 5.0.3(1), and Python 3.7.3—without any special configuration in hardware or software. Except as otherwise noted, (i) all pipelines are set to (initially) read from and (finally) write to the file-system, (ii) `curl` fetches data from a different physical host on the same network connected by 1Gbps links.

Parallelism The level of parallelism is configured using PaSh's `--width=W`, a flag that determines the number of times that PaSh applies its transformations and the resulting width of the data-parallel DFG. PaSh does not control a script's initial parallelism (*e.g.*, a command could spawn 10 processes), and thus the resulting scripts often reach maximum parallelization benefits with a value of `W` smaller than the physical cores of our hardware setup (in our case 64).

6.1 Common Unix One-liners

We first evaluate PaSh on a set of popular, common, and classic Unix pipeline patterns [3, 4, 49]. The goal is to evaluate performance benefits due to PaSh's (i) DFG transformations alone, including how `--width` affects speedup, and (ii) runtime primitives, showing results for several configurations of the runtime lattice (Fig. 9).

Programs Tab. 2 summarizes the first collection of programs. NFA-Regex is centered around an expensive NFA-based backtracking expression and all of its commands are in Ⓢ. Sort is a short script centered around a Ⓟ command. Wf and Top-n are based on McIlroy's classic word-counting program [4]; they use sorting, rather than tabulation, to identify high-frequency terms in a corpus. Spell, based on the original `spell` developed by Johnson [3], is another Unix classic: after some preprocessing, it makes clever use of `comm` to report words not in a dictionary. Shortest-scripts extracts the 15 shortest scripts in the user's `PATH`, using the `file` utility and a higher-order `wc` via `xargs` [49, pg. 7]. Diff and Set-diff compare streams via a `diff` (in Ⓝ, non-parallelizable) and `comm` (in Ⓟ), respectively. Sort-sort uses consecutive Ⓟ commands without interleaving them with commands that condense their input size (*e.g.*, `uniq`). Finally, Bi-grams replicates and shifts a stream by one entry to calculate bigrams.

Results Fig. 10 presents PaSh's speedup as a function of `width`=2–64. Average speedups of the optimized PaSh, *i.e.*, with `eager` and `split` enabled, for `width`={2, 4, 8, 16, 32, 64} are {1.97, 3.5, 5.78, 8.83, 10.96, 13.47}×, respectively. For *No-Eager*, *i.e.*, PaSh' transformations without its runtime support, speedups drop to 1.63, 2.54, 3.86, 5.93, 7.46, 9.35×.

Plots do not include lines for configurations that lead to identical parallel programs. There are two types of such cases. In the first, the *Pash* (blue) and *Pash-w/o-split* (red, hidden) lines are identical for scripts that PaSh does not add `split` since the width of the DFG is constant; conversely, when both lines are shown (*e.g.*, Spell, Bi-grams, and Sort), PaSh has added `split`s due to changes in the DFG width (e.g. due to a Ⓝ command). In the second type, *Pash w/o Split* (red) is identical to *No-Eager* (green, hidden) and *Blocking Eager* (orange, hidden) because the input script features a command in Ⓟ or Ⓝ relatively early. This command requires an aggregator, whose output is of width 1, and beyond

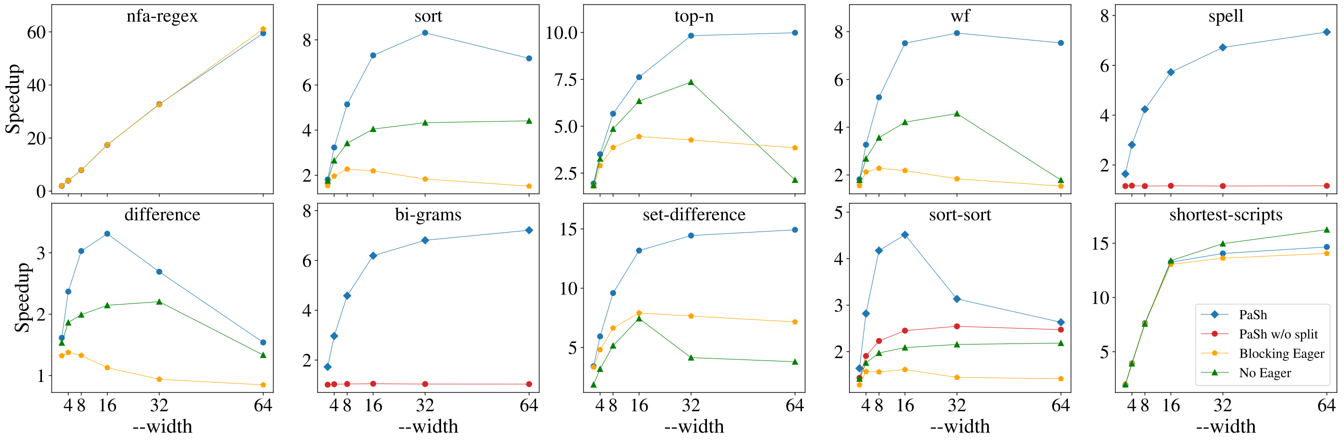


Fig. 10. PaSh's speedup for `width`=2–64. Different configurations per benchmark: (i) PaSh: the complete implementation with `eager` and `split` enabled, (ii) PaSh w/o split: `eager` enabled (no `split`), (iii) Blocking Eager: only blocking `eager` enabled (no `split`), (iv) No Eager: both `eager` and `split` disabled. For some pairs of configurations, PaSh produces identical parallel scripts and thus only one is shown.

which non-`split` configurations are sequential (and see no speedup). Finally, Tab. 2 shows that PaSh's transformation time is negligible, and its COST [32] is 2.

Discussion As expected, scripts with commands only in Ⓢ see linear speedup. PaSh's `split` benefits scripts with Ⓟ or Ⓝ commands, without negative effects on the rest. Its `eager` primitive improves over Eager and Blocking-eager for all scripts. *No Eager* is usually faster than *Blocking Eager* since it allows its producer and consumer to execute in parallel. Sort-sort illustrates the full spectrum of primitives: (i) `eager` offers benefits despite the lack of `split` because it fully parallelizes the first `sort` (ii) *Pash* gets full benefits because `splitting` allows parallelizing the second `sort` too.

As described earlier, PaSh often achieves the maximum possible speedup for a `width` that is lower than the number of available cores—*i.e.*, `width`=16–32 for a 64-core system. This is also because PaSh's runtime primitives spawn new processes—*e.g.*, Sort with `width`=8 spawns 37 processes: 8 `tr`, 8 `sort`, 7 *aggregation*, and 14 `relay` processes.

Take-aways PaSh accelerates scripts by up to 60×, depending on the commands involved. Its runtime constructs improve over the baseline speedup achieved by its transformations.

6.2 Unix50 from Bell Labs

We now turn to a set of Unix pipelines found out in the wild.

Programs In a recent celebration of Unix's 50-year legacy, Bell Labs created 37 challenges [26] solvable by Unix pipelines. The problems were designed to highlight Unix's modular philosophy [30]. We found unofficial solutions to all-but-three problems on GitHub [5], expressed as pipelines with 2–12 stages (avg.: 5.58). They make extensive use of standard commands under a variety of flags, and appear to be written by non-experts (contrary to §6.1, they often use sub-optimal or non-Unix-y constructs). PaSh executes each pipeline as-is, without any modification.

Results Fig. 11 shows the speedup (left) over the sequential runtime (right) for 31 pipelines, with `width`=16 and 10GB inputs. It does not include 3 pipelines that use `head` fairly early thereby finishing execution in under 0.1 seconds. We refer to each pipeline using its x-axis index in Fig. 11. Average speedup is 6.02×, and weighted average (with the absolute times as weights) is 5.75×.

Discussion Most pipelines see significant speedup, except #25-30 that see no speedup because they contain general commands that PaSh cannot parallelize without risking breakage—*e.g.*, `awk` and `sed -d`. A Unix expert would notice that some of them can be replaced with Unix-specific commands—*e.g.*, `awk "{print \$2, \$0}" | sort -nr`, used to sort on the second field can be replaced with a single `sort -nr -k 2` (#26). The limited expressiveness of these commands can be exploited by PaSh—in this specific case, achieving 8.1× speedup (*vs.* the original 1.01×).

For the rest, PaSh's speedup is capped due to a combination of reasons: (i) they contain pure commands that are parallelizable but don't scale linearly, such as `sort` (#5, 6, 7, 8, 9, 19, 20, 21, 23, 24), (ii) they are deep pipelines that already exploit task parallelism (#4, 10, 11, 13, 15, 17, 19, 21, 22), or (iii) they are not CPU-intensive, resulting in pronounced I/O and constant costs (#3, 4, 11, 12, 14, 16, 17, 18, 22).

Take-aways PaSh accelerates unmodified pipelines found in the wild; small tweaks can yield further improvements, showing that PaSh-awareness and scripting expertise can improve results. Furthermore, PaSh does not significantly decelerate non-parallelizable scripts.

6.3 Use Case: NOAA Weather Analysis

We now turn our attention to Fig. 2's script (§2).

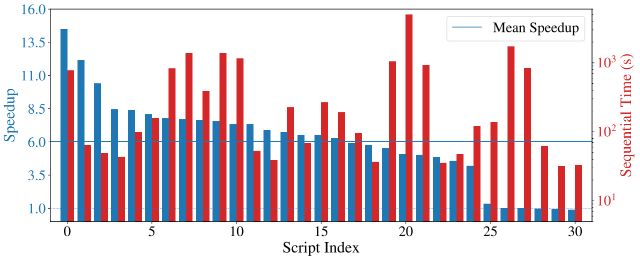


Fig. 11. Unix50 scripts. Speedup (left axis) over sequential execution (right axis) for Unix50 scripts. Parallelism is 16× on 10GB of input data (*Cf.*§6.2). Pipelines are sorted in descending speedup order.

Program This program is inspired by Hadoop's Definitive Guide [51, §2], where it exemplifies a realistic analytics pipeline comprising 3 stages: fetch NOAA data (shell), convert them to a Hadoop-friendly format (shell), and calculate the maximum temperature (Hadoop). While the book focuses only on the last stage, PaSh parallelizes the entire pipeline.

Results The complete pipeline executes in 44m2s for five years (82GB) of data. PaSh with `width=16` leads to 2.52× speedup, with different phases seeing different benefits: 2.04× (vs. 33m58s) for all the pre-processing (75% of the total running time) and 12.31× speedup (vs. 10m4s) for computing the max.

Discussion The speedup of the preprocessing phase of the pipeline is bound by the network and IO costs since `curl` downloads 82GB of data. However, in the processing phase (which is CPU-bound) the speedup is 12.31×, much higher than what would be achieved by parallelizing per loop iteration. Similar to Unix50 (§6.2), we found that large pipelines enable significant freedom in terms of expressiveness.

Take-aways PaSh can be applied to programs of notable size and complexity to offer significant acceleration. A broader take-away is that PaSh is also able to extract parallelism from fragments that are not purely compute-intensive, *i.e.*, the usual focus of conventional parallelization systems.

6.4 Use Case: Wikipedia Web Indexing

We now apply PaSh to a large web-indexing script.

Program This script reads a file containing Wikipedia URLs, downloads the pages, extracts the text from HTML, and applies natural-language processing—*e.g.*, trigrams, character conversion, term frequencies—to index it. It totals 34 commands written in multiple programming languages.

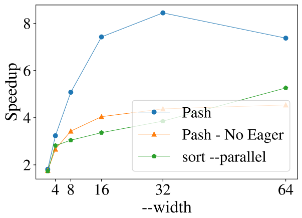
Results The original script takes 191min to execute on 1% of Wikipedia (1.3GB). With `width=16`, PaSh brings it down to 15min (12.7×), with the majority of the speedup coming from the HTML-to-text conversion.

Discussion The original script contains 34 pipeline stages, thus the sequential version already benefits from task-based parallelism. It also uses several utilities not part of the standard POSIX/GNU set—*e.g.*, its `url-extraction` is written in JavaScript and its `word-stemming` is in Python. PaSh can still operate on them as their parallelizability properties—Ⓢ for `url-extract` and `word-stem`—can be trivially described by annotations. Several other stages are in Ⓢ allowing PaSh to achieve benefits by exposing data parallelism.

Take-aways PaSh operates on programs with (annotated) commands outside the POSIX/GNU subsets and leads to notable speedups, even when the original program features significant task-based parallelism.

6.5 Further Micro-benchmarks

As there are no prior systems directly comparable to PaSh, we now draw comparisons with two specialized cases that excel within smaller fragments of PaSh's proposed domain.



Parallel Sort We first compare a GNU `sort` parallelized by PaSh (S_p) *vs.* the same `sort` with the `--parallel` flag set (S_g).[3] While `--parallel` is not a general solution, the comparison serves to establish a baseline for PaSh. S_g's parallelism is configured to 2× that of S_p's `--width` (*i.e.*, the rightmost plot point for S_g is for `--parallelism=128`), to account for PaSh's additional runtime processes.

A few points are worth noting. S_p without `eager` performs comparably to S_g, and with `eager` it outperforms S_g (∼ 2×); this is because `eager` adds intermediate buffers that ensure CPU utilization is high. S_g indicates that `sort`'s scalability is inherently limited (*i.e.*, due to `sort`, not PaSh); this is why all scripts that contain `sort` (*e.g.*, §6.1–6.4) are capped at 8× speedup. The comparison also shows PaSh's benefits to command developers: a low-effort parallelizability annotation achieves better scalability than a custom flag (and underlying parallel implementation) manually added by developers.

GNU Parallel We compare PaSh to `parallel` (v.20160422), a GNU utility for running other commands in parallel [48], on a small bio-informatics script. Sequential execution takes 554.8s *vs.* PaSh's 128.5s (4.3×), with most of the overhead coming from a single command—`cutadapt`.

There are a few possible ways users might attempt to use GNU `parallel` on this program. They could use it on the bottleneck stage, assuming they can deduce it, bringing execution down to 304.4s (1.8× speedup). Alternatively, they could (incorrectly) sprinkle `parallel` across the entire program. This would lead to 3.2× performance improvements but incorrect results with respect to the sequential execution—with 92% of the output showing a difference between sequential and parallel execution. PaSh's conservative program transformations are not applied in program fragments with unclear parallelizability properties.

---

[3] Both `sort`s use the same buffer size internally [40].

7 Related Work

Existing techniques for exploiting parallelism are not directly comparable to PaSh, either because they require significantly more *user* effort (see (§1) for the distinction between users and developers); or are too specialized, targeting narrow domains or custom programming abstractions.

Parallel Shell Scripting Utilities exposing parallelism on modern Unixes—*e.g.*, `qsub` [13], SLURM [52], `parallel` [48]—are limited to embarrassingly parallel (and short) programs and are predicated upon explicit and careful user invocation: users have to navigate through a vast array of different configurations, flags, and modes of invocation to achieve parallelization without jeopardizing correctness. For example, `parallel` contains flags such as `--skip-first-line`, `-trim`, and `--xargs`, and introduces (and depends on) other programs with complex semantics, such as ones for SQL querying and CSV parsing. In contrast, PaSh manages to parallelize large scripts correctly with minimal-to-zero user effort.

Several shells [10, 29, 45] add primitives for non-linear pipe topologies—some of which target parallelism. Here too, however, users are expected to manually rewrite scripts to exploit these new primitives, contrary to PaSh.

Recently, Smoosh [16] argued for making concurrency explicit via shell constructs. The argument is dissimilar from PaSh's, which argues for mostly automated (and correct) parallelization—hence *light-touch* parallel scripting.

Developed independently and at the same time with PaSh, POSH [41] is a framework for running shell scripts over remote storage by offloading the I/O-intensive portions to proxy servers closer to the data. POSH focuses on co-locating distributed operation, and introduces a novel scheduling algorithm. PaSh instead focuses on the parallelization of CPU-intensive scripts, by transforming them to DFGs, applying transformations, and then transforming them back to parallel shell scripts augmented with PaSh's runtime primitives to be executed on a standard shell.

Low-level Parallelization Instruction-level parallelization has a long history, starting from explicit `DOALL` and `DOACROSS` annotations [7, 27] and continuing with compilers that attempt to automatically extract parallelism [18, 37]. These systems operate at a lower level than PaSh (*e.g.*, that of instructions or loops rather than the boundaries of programs that are part of a script), within a single-language or single-target environments, and require source modifications.

More recent work focuses on extracting parallelism from domain-specific programming models [12, 15, 25] and interactive parallelization tools [22, 23]. These tools simplify the expression of parallelism, but still require significant user involvement in discovering and exposing parallelism.

Correct Parallelization of Dataflow Graphs The DFG is a prevalent model in several areas of data processing (including batch- [9, 53] and stream-processing [8, 34]). Despite its popularity, most systems perform optimizations that do not preserve semantics, introducing subtle erroneous behaviors. Recent work [19, 28, 43] attempts to address this issue by performing optimizations only in cases where correctness is preserved. PaSh draws inspiration from these efforts, as it attempts transformations that maintain the program's correctness with respect to the sequential execution. Its DFG model, however, is different and captures ordering constraints. This is due to the intricacies of the Unix model—*e.g.*, streams, argument processing, and concatenation operators.

Parallel Userspace Environments By focusing on simplifying the development of distributed programs, a plethora of environments inadvertently assist in the construction of parallel software. Such systems [1, 33, 36, 39] or languages [11, 24, 44, 50] hide many of the challenges of dealing with concurrency as long as developers leverage the provided abstractions—which are strongly coupled to the underlying operating or runtime system. Even shell-oriented efforts such as Plan9's `rc` are not backward-compatible with the Unix shell, and often focus primarily on hiding the existence of a network rather than automating parallel processing.

Parallel Frameworks Several frameworks [2, 6, 14, 47] offer fully automated parallelism as long as special primitives are used—*e.g.*, map-reduce-style primitives for Phoenix [47]. These primitives make strong assumptions about the nature of the computation—*e.g.*, strongly-eventual commutative functions that can proceed in parallel. By targeting specific classes of computation (*viz.* PaSh's parallelizability), they are significantly optimized for their target domains. PaSh chooses a more general approach: it does not require setting up a new framework for each new class of computation used, nor rewriting different parts of the computation using a different set of abstractions provided by each framework.

Dryad [21] is a distributed system for dataflow graphs. Dryad offers a scripting language, Nebula, that allows using shell commands such as `grep` or `sed` in place of individual dataflow nodes. The main difference with PaSh is that in Dryad the programmer needs to explicitly express the dataflow graph, which is then executed in a distributed fashion, whereas Pash automatically parallelizes a given shell script by producing a parallel script that runs on an unmodified shell of choice.

8 Conclusion

Shell programs are ubiquitous, utilize programs from a plethora of programming languages, and spend a significant fraction of their time interacting with the broader environment to download, extract, and process data—falling outside the focus of conventional parallelization systems. This paper presents PaSh, a system that allows shell users to parallelize shell programs mostly automatically. PaSh can be viewed as (i) a source-to-source compiler that transforms scripts to DFGs, parallelizes them, and transforms them back to scripts; coupled with (ii) a runtime component that addresses several practical challenges related to performance and correctness. PaSh's extensive evaluation over 44 unmodified Unix scripts demonstrates non-trivial speedups (0.89–61.1×, avg: 6.7×).

PaSh's implementation, as well as all the example code and benchmarks presented in this paper, are all open source and available for download: github.com/andromeda/pash.

Acknowledgments

We want to thank André DeHon, Ben Karel, Caleb Stanford, Thurston Dang, Jean-Sébastien Légaré, and Nick Roessler. This material is based upon work supported by DARPA contract no. HR00112020013 and NSF award CCF 1763514.

References

[1] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[2] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: a c++ template library for high performance stream parallel processing. *The International Journal of High Performance Computing Applications*, 31(5):391–404, 2017.

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

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

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

[6] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[7] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

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

[10] Tom Duff. Rc—a shell for plan 9 and unix systems. *AUUGN*, 12(1):75, 1990.

[11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[12] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[13] 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.

[14] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 291–303, New York, NY, USA, 2002. Association for Computing Machinery.

[15] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002.

[16] Michael Greenberg. The posix shell is an interactive dsl for concurrency. https://cs.pomona.edu/~michael/papers/dsldi2018.pdf, 2018.

[17] The Open Group. Posix. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018. [Online; accessed November 22, 2019].

[18] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[19] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

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

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[22] Makoto Ishihara, Hiroki Honda, and Mitsuhisa Sato. Development and implementation of an interactive parallelization assistance tool for openmp: ipat/omp. *IEICE transactions on information and systems*, 89(2):399–407, 2006.

[23] Ken Kennedy, Kathryn S McKinley, and C-W Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[24] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[25] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.

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

[27] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[28] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 670–685, New York, NY, USA, 2019. ACM.

[29] Chris McDonald and Trevor I Dix. Support for graphs of processes in a command interpreter. *Software: Practice and Experience*, 18(10):1011–1016, 1988.

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

[31] Peter M McIlroy, Keith Bostic, and M Douglas McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[32] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what COST? 15:241–299, 2015.

[33] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[35] National Oceanic and Atmospheric Administration. National climatic data center. https://www.ncdc.noaa.gov/, 2017.

[36] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[37] David A Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, 1993.

[38] Davide Pasetto and Albert Akhriev. A comparative study of parallel sort algorithms. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 203–204, 2011.

[39] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[40] Pixelbeat. Answer to: Sort –parallel isn't parallelizing, 2015. Accessed: 2020-04-14.

[41] 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.

[42] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *SIGOPS Oper. Syst. Rev.*, 7(4):27–, January 1973.

[43] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb 2015.

[44] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 15–26, New York, NY, USA, 2005. ACM.

[45] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[46] Richard M Stallman and Roland McGrath. Gnu make—a program for directing recompilation. https://www.gnu.org/software/make/manual/make.pdf, 1991.

[47] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

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

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

[50] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[51] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[52] 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.

[53] 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.