PASH: Light-touch Data-Parallel Shell Processing

Nikos Vasilakis* Konstantinos Kallas† Konstantinos Mamouras°
Achilleas Benetopoulos◇ Lazar Cvetkovich‡
*MIT, CSAIL †University of Pennsylvania °Rice University ◇Unaffiliated ††University of Belgrade

Abstract

This paper presents PASH, a shell aimed at parallelizing POSIX shell scripts through a combination of program transformations and runtime primitives. Given a script, PASH converts it to a data-flow graph, performs a series of analyses that aim at exposing parallelism while preserving its sequential semantics, and then converts dataflow graph back into a script that uses POSIX constructs to explicitly guide parallelism. A set of lightweight annotations allow 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, informs PASH's annotation language and results in its equivalent of a data-parallel standard library. Finally, PASH's UNIX-aware runtime primitives address several practical issues related to performance and correctness. PASH's extensive evaluation, combining several classic and modern Unix scripts, shows significant benefits across several axes—with order-of-magnitude performance improvements 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 [29], makes the shell the primary choice for specifying succinct and simple pipelines 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` [37, 30, 43].

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 manual tools such as GNU `parallel` [45], `ts` [20], `qsub` [13], SLURM [49]; 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`, `for`) 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 whose domain of expertise falls outside computing—routinely writing shell scripts to accomplish their tasks—are left without options.

To address this challenge, we develop PASH, a shell that extracts data-parallelism from POSIX shell scripts. PASH benefits both programmer groups, with a particular emphasis on users. Command developers are provided with a set of abstractions, akin to lightweight type annotations, for expressing the parallelizability properties of their commands: rather than providing a command's full observable behavior, these annotations focus primarily on their interaction with state. Shell users, on the other hand, are provided with full automation: PASH extracts parallelism latent in their scripts by analyzing their parallelizability properties. 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 semantics 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.[1]

[1] In fact, the scope of the study is broader, incorporating also commands

These two components are finally paired 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 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.

We evaluate PASH using (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) a series of micro-benchmarks inspired by (and comparing with) possible alternatives. The results show significant advantages in correctness, manual effort, and performance—with an order-of-magnitude performance improvement due to PASH's transformations and runtime primitives. Improvements depend on several factors such as the execution time, the pipeline depth, and the parallelizability classes in which its commands fall.

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 a large set of shell commands and introduces a lightweight annotation language for commands that can be executed in a data-parallel manner.
- §4 presents a dataflow graph model for encoding shell scripts, and a set of parallelization transformations that focus on preserving the semantics of the sequential program.
- §5 details PASH's implementation, discussing several challenges related to PASH's translation, optimization, and runtime components and their solutions.

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 ten years. As the National Oceanic and Atmospheric Administration (NOAA) has made historic temperature data publicly available [34], answering this question is only a matter of a simple data-processing pipeline.

The shell script in Fig. 1 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

outside POSIX and GNU Coreutils such as `curl` and `pandoc`.

```
base="ftp://ftp.ncdc.noaa.gov/pub/data/noaa";
for y in {2015..2020}; 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. 1: Calculating maximum temperatures per year. The script downloads daily temperatures recorded across the U.S. for the years 2015–2020 and extracts the maximum for every year.

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 required to write this script is astonishingly low:[2] its data-processing core amounts to 12 stages and, when expressed as a single line, is only 165 characters long. Its succinctness is deceiving—this program is no toy: a Java program implementing only the last four pipeline stages requires 137 LoC [48, §2.1]. To enable such a succinct program composition, UNIX incorporates several features.

UNIX Features Composition in UNIX is primarily achieved through pipes (`|`), a construct that allows for task-parallel execution of two commands by connecting them with a character stream. These streams are contiguous character lines separated by newline characters (NL), which delineate individual stream elements. For example, the first `grep` is given lines that contain file identifiers, of which it only outputs lines that contain `gz`, which are in turn 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 the commands used in practice come with the system—*e.g.*, commands defined by the POSIX standard or ones part of the GNU Coreutils. 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 the call to `sort` and `head` with one to `./avg.py` to get the average or standard deviation rather than the maximum. The pipeline will still work, as long as `./avg.py` con-

[2] Some effort is required to understand NOAA's weather format, but this is true for any program processing any dataset.

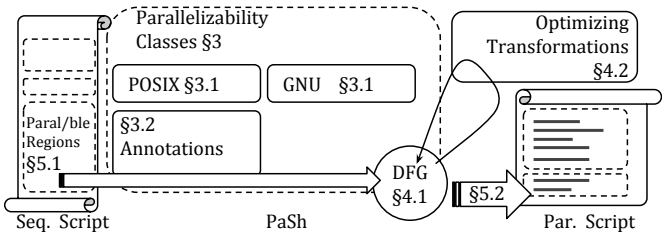


Fig. 2: PASH overview. A high-level schematic outline of PASH.

forms to the interface outlined earlier.

2.2 Parallelization Challenges

While these features aid in development effort economy through powerful program composition, they complicate shell script parallelization, which even for simple scripts such as the one in Fig. 1 create several challenges.

Commands In contrast to restricted programming frameworks that enable parallelization by supporting a few carefully-designed primitives [14, 6, 9, 50], the UNIX shell provides an unprecedented number and variety of composable commands. To be parallelized, each command may require special analysis and treatment—in Fig. 1, `grep`'s results can be combined by simple concatenation, while `sort`'s results would require careful merging. 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. 1's `cut` is called twice with two different sets of flags.

Scripts Another challenge is due to the language of the POSIX shell. First, it contains constructs that enforce sequential execution; Fig.1'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 `&`, `for`, and `|` constructs. Even though Fig. 1'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 and induce as much parallelism as it can without breaking semantics.

Implementation Independently of the semantics of individual commands and the shell language, the UNIX environment has its own set of quirks. Thus, any attempt to orchestrate parallel execution will have to take care of several challenges related to task parallelism, deadlock prevention, and runtime performance. For example, forked processes piping their combined results to Fig. 1's `head` might 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.

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

Fig. 3: PASH's 2×-parallel output for fragment of Fig. 1. PASH orchestrates the script's parallel execution through named pipes, parallel operators and careful placement of combiners.

2.3 PASH Design Overview

At a high level, PASH takes as input a POSIX shell script like the one in Fig. 1 and outputs a new script. The new script may incorporate parallelism, and is handed off to the user's original shell interpreter for execution. To expose and exploit data parallelism, PASH analyzes individual program fragments, addressing the challenges mentioned earlier.

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*. Rather than describing a command's full observable behavior, these classes focus on information that is important for data parallelism. To allow commands that fall outside of the standard set to leverage its transformations, PASH defines a light *annotation language* for describing a command's parallelizability class. Annotations are expressed once per command rather than once per script and are aimed 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 *parallelizable regions* that contain commands that are candidates for parallelization. This analysis is guided by the script structure: some constructs (*e.g.*, `&`, `|`) expose task parallelism; others (*e.g.*, `&&`, `||`) enforce synchronization. PASH then converts each parallelizable region to a *dataflow graph* (DFG), a flexible representation that enables a series of local transformations to expose data parallelism, converting the graph into its parallel equivalent. 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.

Implementation PASH addresses several practical challenges through a set of constructs it provides—*e.g.*, a set of modular components for augmenting command composition. It also provides a small library of efficient data aggregators for 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.

Fig 3 shows a fragment of PASH's output for the script in Fig. 1. The next few sections (§3–5) discuss the details.

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%)

3 Parallelizability Classes

PASH aims at parallelizing data-parallel commands, *i.e.*, commands that can process their input in parallel, and encodes their characteristics by assigning them to *parallelizability classes*. A class represents the level of synchronization required by copies of a command executing in parallel, capturing only characteristics that are important for its parallel execution. 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 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 need 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 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 turns out to be useful for cases with very large stream elements (*i.e.*, long lines) such the `.fastq` format used in bioinformatics pipelines.

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 a *fold* 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 [39]. 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 four crucial concerns: (i) a command's parallelizability class, (ii) a command's inputs and outputs, (ii) for commands that support multiple inputs, the characteristics of input consumption, and (iii) for commands with multiple flags (options), the handling of competing sets of flags. As the first concern was discussed extensively in the previous section, we now focus on the latter three.

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 to correctly connect them with the inputs of the next. 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 parallel version of the expression `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 to control 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.

As parallelizability classes form a hierarchy from most parallelizable to least parallelizable (§3.1), a command is classified by the class of its least parallelizable flag. For example, if a custom `trace-sort` command is invoked with flags `r`, `n`, and `k2` that are in Ⓟ and a flag `d` in Ⓔ that writes debugging output to a file, it ends up in Ⓔ.

Example Annotations The parallelizability properties of each command are written in an annotation record that contains a set of clauses, each one identified by a predicate on the command options. Each clause maps to an assignment for the command, indicating its class and the sequence of its inputs and outputs. Annotations treat flag arguments differently from file arguments, by checking if they start with a `-`. The complete annotation grammar is shown in Appendix A.

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). Its annotation record is shown below:

```
comm {
| -1 /\ -3 => (S, [arg[1]], [stdout])
| -2 /\ -3 => (S, [arg[0]], [stdout])
| _  => (P, [arg[0], arg[1]], [stdout])
}
```

The first two clauses describe `comm` invocations with options `-1` (or `-2`) and `-3`. As the first (second) and third columns are suppressed, `comm` can be considered 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 third clause is the general case, which includes `comm` invocations without any options, and is captured by the third clause. There, `comm` is classified as Ⓟ as it needs to read both inputs completely before it can output its results. Its inputs are its first and second file arguments (*i.e.*, excluding any options), and its output is produced in `stdout`.

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. If these invariants are met, after DFG analysis (§4) PASH can replace occurrences of these commands with their *map* and *aggregate* equivalents, enabling their full parallelization.

PASH defines aggregators for many Ⓟ POSIX and GNU commands, which operate as both PASH's standard library and an exemplar for community attempts to tackle other commands.

4 Dataflow Graph Model

PASH's core is an abstract dataflow graph (DFG) model that is used as the intermediate representation on which PASH performs optimizations. PASH first lifts sections of the input script to the DFG representation, it performs transformations to expose parallelism, and then instantiates each DFG back to a parallel shell script. A fundamental difference with other DFG models is that PASH's DFG encodes the order in which a node reads its inputs, which in turn enables a set of graph transformations that can be iteratively applied to expose parallelization opportunities for Ⓢ and Ⓟ commands. In Section 4.1 we define the DFG model and describe the correspondence with shell scripts, and in Section 4.2 we describe the graph transformations that PASH performs.

4.1 Definitions

The two main shell abstractions are (i) streams, *i.e.*, files or pipes, that contain data, and (ii) commands that communicate through these streams. PASH's DFG model represents commands as nodes and streams as edges.

We first introduce basic notation used in 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 $f : [D^*] \to [D^*]$, 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`, 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.

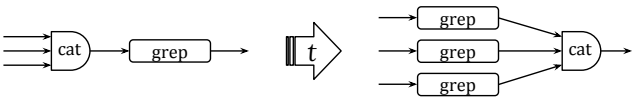


Fig. 4: Stateless parallelization transformation. The `cat` node is commuted with the stateless node to utilize available data parallelism.

4.2 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 As previously noted (§3.1), stateless commands such as `tr` operate independently on individual elements of the stream (characters, lines, or files) without maintaining any state. 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:

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

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 concatenation of 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

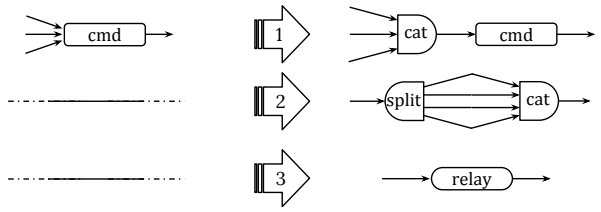


Fig. 5: Auxiliary transformations. These augment the DFG with `cat`, `split`, and `relay` nodes.

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. 4). 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.

T can be extended straightforwardly to nodes $v \in$ Ⓟ, implemented by a map-aggregate pair (m, agg) as $v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow agg(m(x_1, s) \cdot m(x_2, s) \cdots m(x_n, s), s)$. 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 Figure 5. 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 (§5.2).

5 From Scripts to Graphs and Back Again

This section describes key points in the implementation of PaSh. It first details how its front-end translates a script to the DFG model defined in §4 by identifying and transforming parallelizable regions (§5.1). It then describes its back-end, responsible for generating the parallel script (§5.2).

5.1 PaSh Front-End

To optimize a script, PaSh translates it to a DFG (§4). As there are several program fragments that cannot be parallelized—*e.g.*, commands connected with `&&`, §2—PaSh introduces the notion of *parallelizable program regions* and a translation pass for converting them to DFGs.

Parallelizable Regions Parallelizable regions are program sub-expressions that can be parallelized safely, *i.e.*, without breaking the program. The search for these regions is guided by the shell language and the structure of a particular program. These contain information about two types of components: fragments that can be executed independently and barriers that are natural synchronization points. Consider this 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. Both `cat f1 f2 | grep "foo" > f3` and `sort f3` are thus parallelizable regions, each not extending beyond `&&`—more precisely, they are maximal.

Intuitively, parallelizable regions correspond to sub-expressions of the program that would be allowed to execute independently by different processes in the POSIX standard [17]. Larger parallelizable regions can be composed from smaller ones using the pipe operator (`|`) and the parallel-composition operator (`&`). Conversely, all other operators, such as the sequential composition operator (`;`) and the logical operators (`&&`, `||`), represent barrier constructs that do not allow parallelizable regions to permeate through.

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 parallelizable regions bottom-up, translating their independent components to DFG nodes until a barrier construct is reached. All subtrees not translatable to DFGs are kept as they are. The output of the translation pass is the original AST where parallelizable regions have been replaced with DFGs and calls to PaSh's runtime primitives (§5.2).

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 parallelizable 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 includes 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, for example, an environment variable passed as an argument to a command does not change its parallelizability class.

5.2 PaSh Back End

After translating the parallelizable regions of the input script to DFGs and applying optimization passes (§4.2), 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. This section describes

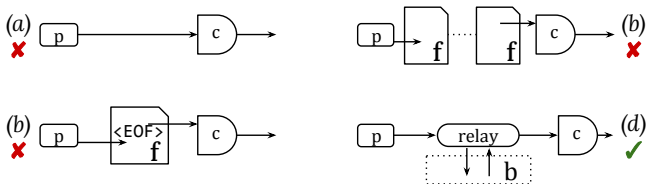


Fig. 6: Eager primitive. Addressing intermediary laziness is challenging: (a) FIFOs are blocking; (b) files alone introduce race conditions between producer/consumer; (c) files `wait` inhibit task-based parallelism. Eager `relay` nodes (d) address the challenge while remaining within the PASH model.

technical challenges related to execution of the resulting script and how they are addressed by PASH's custom 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 inhibits opportunities for parallelism, as commands are often blocked when their consumers are not requesting any input. Consider the following script:

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

The `cat` command will try to consume input from `t2` after it completes reading from `t1`. As a result, the second `grep` will remain blocked until the first `grep` completes (Fig. 6a).

One might be tempted to replace FIFOs with files, a central UNIX abstraction, simulating pipes of arbitrary buffering (Fig. 6b). 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. 6c); however, this would insert artificial barriers impeding task-based parallelism and wasting disk.

To address this challenge, PASH inserts and instantiates eager `relay` nodes at these points (Fig. 6d). These nodes feature tight multi-threaded loops that consume input eagerly while attempting to push, forcing upstream nodes to produce output when possible while also preserving task-based parallelism.

Splitting Challenges PASH's optimizer inserts `split` nodes to expose parallelism when parallelizable nodes only have one input (§4.2). 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 solve this, PASH supports two `split` implementations: (i) a general one that can be used with commands that could add or remove lines, and (ii) an optimized one that can be used if the input size is known before its execution. In the former case, `split` first consumes its complete input, counts its lines, and then splits it uniformly across the desired number of outputs. In the latter case, it can be configured to avoid reading all its input, thus exploiting available task-based parallelism.

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[3] 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:

```
mkfifo fifo1 fifo2
cat file-chunk1.txt > fifo1 &
cat file-chunk2.txt > 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 include *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 combiners 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 scale shell scripts out automatically and correctly, using several scripts collected out from the wild along with a few micro-benchmarks for targeted comparisons.

[3] 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 parallelization configurations.

Script	Structure	Input	Seq. Time	#Nodes(16, 64)		Compile Time (16, 64)		Highlights
Grep	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	`sort`ing
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
Grep-light	3 × Ⓢ	100 GB	1m38.212s	49	193	0.031s	0.163s	IO-intensive, computation-light
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 Ⓟ
Diff	2 × Ⓢ, 3 × Ⓟ	10 GB	25m49.097s	125	509	0.186s	2.341s	non-parallelizable `diff`ing
Bi-grams	3 × Ⓢ, 3 × Ⓟ	3 GB	38m9.922s	185	761	0.146s	1.716s	stream shifting and merging
Bi-grams-opt	3 × Ⓢ, Ⓟ	3 GB	38m21.501s	63	255	0.117s	1.482s	optimized version of bigrams
Set-diff	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 Ⓟ

Highlights PASH accelerates $(1.92 - 61.1\times)$ almost all scripts (> 50, with > 200 commands). From the rest, only two see a slowdown, only because PASH's setup ($\sim 1s$) is higher than their runtime ($\sim 0.4s$). PASH's optimized primitives offer benefits even against manually parallelized scripts. In two large cases, tasks such as data download, extraction, and preprocessing—often implemented using shell scripts and outside the focus of specialized parallelization frameworks—take comparable time to the execution of the main computation, and can be significantly accelerated by PASH.

The vast majority requires no effort other than invoking PASH; the annotation for the remaining 6 commands ($< 3\%$) amounts to a single record. In terms of correctness, PASH's results over multi-GB inputs are identical to the sequential for all benchmarks. Scripts feature ample opportunities for breaking semantics, 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.

A Note on the Parallelism Factor The discussion of the results often mentions the parallelism factor, which refers to the PASH *parallelism configuration* rather than the resulting parallelization. In fact, for small parallelism configurations PASH often achieves higher parallelization due to the creation of multiple processes—aggregators, splitters, relays *etc.*

6.1 Common UNIX One-liners

We evaluate PASH under multiple configurations on popular, common, or classic UNIX pipeline patterns [3, 4, 46].

Programs Tab. 2 summarizes the collection of programs shown in Fig. 7, which in turn reports on their performance characteristics. Grep and Grep-light are in Ⓢ, with the former centered around an expensive DFA-based backtracking expression. 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` [46, 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, the two Bi-grams replicate and shift a stream by one entry to calculate bigrams, but Bi-gram-opt's shift, `sort`, and `uniq` are merged to one command to use a more efficient aggregator.

Results Fig. 7 presents PASH's speedup as a function of parallelism (2–64×). (Bi-grams-opt is shown as the `opt` line in Fig. 7's Bi-grams.) Average speedups of the best PASH configuration, *i.e.*, `eager` enabled with `split` when it provides benefit, for {2, 4, 8, 16, 32, 64}×-parallelism are 1.97, 3.5, 5.78, 8.83, 10.96, and 13.47, respectively.

Tab. 2 shows that PASH's compilation time is negligible. It also contains Grep-light (not in Fig. 7), designed to show that PASH does not slow down IO-intensive scripts; on the contrary, PASH achieves a 1.5-2.5× speedup for Grep-light. PASH achieves a COST [31] of 2 for all benchmarks.

Discussion PASH achieves near-linear speedup on low-parallelism configurations. This is due to a high degree (deep trees) of task parallelism from PASH's multiple constructs—*e.g.*, *aggregations* and `relay`s—evident in all plots that contain Ⓟ stages. For example, Sort in 8× spawns 37 nodes: 8 `tr` nodes, 8 `sort` nodes, 7 *aggregation* nodes, and 14 `relay` nodes. In effect, PASH hits the maximum number of parallel processes (and thus achieves optimal performance, *i.e.*, one

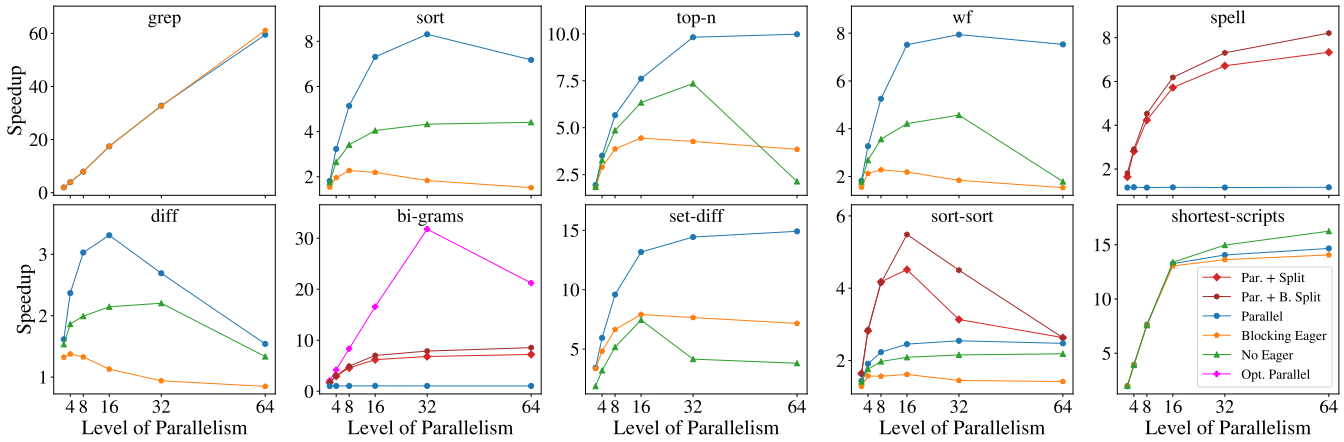


Fig. 7: PASH's speedup for 2–64×-parallelism. Different configurations per benchmark: (i) Par+Split: `eager` and general `split` enabled, (ii) Par+B.Split: `eager` and input-aware `split` enabled, (iii) Parallel: `eager` enabled (no `split`), (v) Blocking Eager, only blocking `eager` enabled (no `split`), (iv) No Eager: both `eager` and `split` disabled, (vi) Opt. Parallel, an optimized version of `bigrams` (*Cf.*§6.1). Only relevant configurations are shown—*e.g.*, `grep` and `sort` do not benefit from `split`; `spell` and `bi-grams` do not see benefits without `split`.

without unnecessary process spawns and context switches) significantly earlier than the purported parallelism—16 − 32× for a 64-core system. Asking for parallelism beyond that point degrades speedup, as overheads still increase but no further parallelism is available—as is the case, *e.g.*, with Sort-sort for 32× and 64× which has 314 and 634 nodes, respectively.

No Eager and Blocking Eager depict configurations without PASH's `eager` optimization and without `split`. As described in Section 5.2, these configurations perform worse than PASH with `eager` in all cases (except for a negligible difference in shortest-scripts). Adding `split` on top of that (Par + Split, Par + B. Split) further improves speedup for some scripts that have Ⓟ or Ⓝ commands, and for the rest it does not affect performance. Finally, the optimized bi-grams script achieves significantly more benefits from PASH than its unoptimized counterpart, showing that familiarity with PASH's parallelizability classes could go a long way in terms of performance.

Take-aways PASH accelerates scripts by 4–60×, 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 evaluate PASH on an existing set of UNIX pipelines found in the wild and written by non-experts.

Programs In a recent celebration of UNIX's 50-year legacy, Bell Labs created 37 challenges [25] solvable by means of composing UNIX pipelines. We found unofficial solutions to all-but-three problems on GitHub [5], expressed as pipelines with 2–12 stages (avg.: 5.58). We consider this a good set of benchmarks because (i) the problems were designed to highlight UNIX's modular philosophy [29] and make extensive use of standard commands under a variety of flags, and (ii) the solutions were written by non-experts (contrary to §6.1). PASH executes each pipeline as-is; if an obvious fix improves performance, we report the speedup of both versions.

Results PASH's parallelism is set to 16×, as many of these pipelines are relatively long (§6.1, Discussion). Their input was generated by multiplying their original input several times, enough to reach 10GB. Fig. 8 shows the speedup (left) over the sequential runtime (right) for all 34 pipelines. We use the pipeline index to refer to them in both plots and discussion. Average speedup is 5.49×, median is 6.07×, and weighted average (with the absolute times as weights) is 5.75×.

Discussion Most pipelines see significant acceleration, except 13, 24, 25, 26, 29, 30 that see no speedup and 2, 19, 31 that see a slowdown. The first group contains commands which PASH cannot parallelize without risking breakage—*e.g.*, `awk` and `sed -d`. Note however, that some of the pipelines contain more general commands that cannot be parallelized since they were not written with that goal in mind; a common example is the use of `awk` for reordering or removing columns of the input. In particular, replacing an `awk` that was just used to sort on the second field `awk "print $2, $0" | sort -nr` with a single `sort -nr -k 2` in the 13th pipeline allows PASH to achieve 8.1× speedup (in contrast to the original 1.01×).

The second group contains `head`, thus practically processing only one line (regardless of the full input) and completing within 10*ms*. PASH's slowdown is due to constant setup costs of pipes and processes but execution still remains under 1*s*.

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` (0, 1, 3, 15, 16, 18, 20, 27, 28, 33), (ii) some are deep pipelines that already exploit task-based parallelism (7, 8, 9, 10, 11, 12, 14, 27, 28,) and (iii) they are not CPU-intensive, resulting in pronounced IO and constant costs (4, 5, 6, 7, 8, 12, 14, 22, 23).

Take-aways PASH accelerates unmodified pipelines found

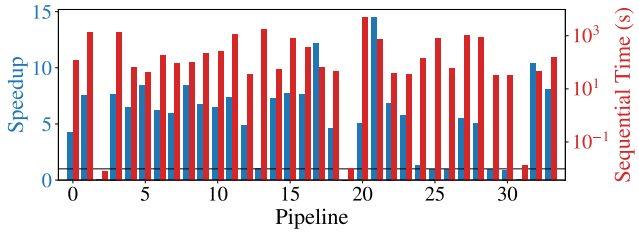


Fig. 8: Unix50 scripts. Speedup (left axis) over sequential execution (right axis) for Unix50 scripts. Parallelism is 16× on 10GB of input data (*Cf.*§6.2).

in the wild; small tweaks can yield further improvements, showing that PASH-awareness and scripting expertise can improve results. Furthermore, PASH does not decelerate non-trivial computations, even when parallelism is impossible.

6.3 Use Case: NOAA Weather Analysis

We now turn our attention to Fig. 1's script (§2).

Program This program is inspired by Hadoop's Definitive Guide [48, Chapter 2], where it exemplifies a realistic analytics pipeline comprising three sub-tasks: fetch data from NOAA (shell), convert them to a Hadoop-friendly format (shell), and calculate the maximum temperature (Hadoop). Only the last one is the real focus of the book, whereas we consider the entire pipeline.

Results The complete pipeline executes in 44m2s for five years (82GB) of data. PASH with 2/10× parallelism leads to 1.86/2.44× speedup, with different phases seeing different benefits: 1.77/1.99× (vs. 33m58s) for all the pre-processing and 2.30/10.79× speedup (vs. 10m4s) for computing the max. (For why speedup is higher than parallelism, see §6.1.)

Discussion Similar to Unix50 (§6.2), we found that large pipelines enable significant freedom in terms of expressiveness. A few stages of the original script were expressed in a single `awk`; a simpler (but longer) pipeline that leverages UNIX built-ins turns out to be trivially parallelizable.

Take-aways PASH can be applied to programs of notable size and input 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 regions, which are the 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. In total, it contains 34 commands written in multiple programming languages.

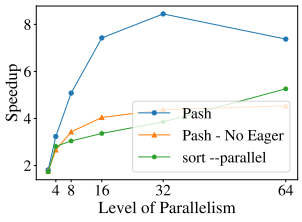
Results The original script takes 191min to execute on 1% of Wikipedia (1.3GB). With 2/16×-parallelism, PASH brings it down to 97/15min (1.97/12.7×), with the majority speedup coming from the HTML-to-text conversion.

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-extract`ion is written in JavaScript and its `word-stem`ming 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

This section attempts to characterize PASH's limits by comparing it with similar alternatives. As there are no prior systems directly comparable to PASH, we draw comparisons with a series of specialized ones that excel within smaller fragments of PASH's proposed domain.



Parallel Sort We first compare a single GNU `sort` optimized by PASH (S_{PaSh}) versus the same `sort` with the `--parallel` flag set (S_{GNU}). While `--parallel` is not a general solution, the comparison serves to establish a baseline for PASH. S_{GNU}'s parallelism is configured to 2× that of S_{PaSh}'s (*i.e.*, the rightmost plot point for S_{GNU} is for `--parallelism=127`), to account for PASH's additional merge processes.

A few points are worth noting. S_{PaSh} without `eager` performs comparably to S_{GNU}, and with `eager` it outperforms S_{GNU} (∼2×); this is because `eager` adds intermediate buffers between merge phases. S_{GNU} indicates that `sort`'s scalability is inherently limited (*i.e.*, due to `sort` and not PASH); this is why all scripts that contain `sort` (*e.g.*, §6.1–6.4) do not go above 8×. Finally, the comparison also shows PASH's benefits to command developers: a low-effort parallelizability annotation allows similar or better scalability than a custom flag added by developers.

GNU Parallel We compare PASH to GNU `parallel` (v.20160422), a GNU utility for running other commands in parallel [45], on a small bio-informatics script. Sequential execution takes 554.8s *vs.* PASH's 128.5s (4.3×). We note that this pipeline is harsh for PASH, in that most of the

overhead comes from a single command—masking PASH's improvements in other parts of the pipeline.

There are a few possible ways one might attempt to use GNU `parallel` on this program. They could use it on the 8th stage, assuming they know it is a bottleneck, bringing execution down to 304.4s (1.8× speedup). Alternatively, they could (incorrectly) sprinkle `parallel` across the entire program—a strategy simplified by `parallel`'s `stdin` redirection feature. This would lead to 3.2× performance improvements but severely 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 will not operate on fragments with unclear parallelizability properties.

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 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 [49], GNU `parallel` [45]—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, 28, 42] 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.

Low-level Parallelization Instruction-level parallelization has a long history, starting from explicit `DOALL` and `DOACROSS` annotations [7, 26] and continuing with compilers that attempt to automatically extract parallelism [36, 18]. 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, 24] and interactive parallelization tools [22, 21]. 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, 50] and stream-processing [33, 8]). Despite its popularity, most systems perform optimizations that do not preserve semantics, introducing subtle erroneous behaviors. Recent work [19, 40, 27] 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 [35, 32, 38, 1] or languages [47, 41, 23, 11] 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 [14, 6, 44, 2] offer fully automated parallelism as long as special primitives are used—*e.g.*, map-reduce-style primitives for Phoenix [44]. 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.

8 Conclusion

This paper presented PASH, a shell variant that parallelizes shell programs mostly automatically. PASH's insight is that shell pipelines already express streaming computations that can be automatically distributed. To achieve its goal, PASH decomposes primitives into parallelizability classes, identifies high-parallelizability stages, applies a series of transformations according to a DFG model, and orchestrates the execution of the resulting parallel program. PASH can lead to significant benefits for shell users. Experiments with real programs show substantial speedups and the ability to operate on large input datasets, all with minimal or zero user effort.

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, 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] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

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

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

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

[32] 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.

[33] 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.

[34] National Oceanic and Atmospheric Administration. National climatic data center, 2017.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] Richard M Stallman and Roland McGrath. Gnu make—a program for directing recompilation. 1991.

[44] 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.

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

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

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

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

[49] 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.

[50] 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.

A Annotation Language Grammar

Figure 9 shows the complete grammar of the parallelizability annotation language.

⟨*option*⟩ ::= '-' ⟨*string*⟩

⟨*category*⟩ ::= '`stateless`' | '`pure`' | ...

⟨*maybe-int*⟩ ::= | ⟨*int*⟩

⟨*arg*⟩ ::= '`args[`' ⟨*int*⟩ '`]`'

⟨*args*⟩ ::= ⟨*arg*⟩
| '`args[`' ⟨*maybe-int*⟩ '`:`' ⟨*maybe-int*⟩ '`]`'

⟨*input*⟩ ::= '`stdin`' | ⟨*args*⟩

⟨*inputs*⟩ ::= ⟨*input*⟩
| ⟨*input*⟩ '`,`' ⟨*inputs*⟩

⟨*output*⟩ ::= '`stdout`' | ⟨*arg*⟩

⟨*outputs*⟩ ::= ⟨*output*⟩
| ⟨*output*⟩ '`,`' ⟨*outputs*⟩

⟨*option-pred*⟩ ::= ⟨*option*⟩
| '`value`' ⟨*option*⟩ = ⟨*string*⟩
| '`not`' ⟨*option-pred*⟩
| ⟨*option-pred*⟩ '`or`' ⟨*option-pred*⟩
| ⟨*option-pred*⟩ '`and`' ⟨*option-pred*⟩

⟨*assignment*⟩ ::= '`(`' ⟨*category*⟩, '`[`' ⟨*inputs*⟩ '`]`' '`,`' '`[`' ⟨*output*⟩ '`]`' '`)`'

⟨*predicate*⟩ ::= ⟨*option-pred*⟩ '`=>`' ⟨*assignment*⟩

⟨*pred-list*⟩ ::= '`|`' ⟨*predicate*⟩ ⟨*pred-list*⟩
| '`|`' '`otherwise`' '`=>`' ⟨*assignment*⟩

⟨*command*⟩ ::= ⟨*name*⟩ '`{`' ⟨*pred-list*⟩ '`}`'

⟨*command-list*⟩ ::= ⟨*command*⟩
| ⟨*command*⟩ ⟨*command-list*⟩

Fig. 9: Parallelizability description language. The DSL captures important