PaSh: Light-touch Data-Parallel Shell Processing

Nikos Vasilakis*
MIT
nikos@vasilak.is

Konstantinos Kallas*
University of Pennsylvania
kallas@seas.upenn.edu

Konstantinos Mamouras
Rice University
mamouras@rice.edu

Achilles Benetopoulos
Unaffiliated
abenetopoulos@gmail.com

Lazar Cvetković
University of Belgrade
cl203023m@student.etf.bg.ac.rs

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.

CCS Concepts • **Software and its engineering → Compilers**; **Massively parallel systems**; *Scripting languages.*

Keywords Automatic Parallelization, Shell, Pipelines, Source-to-source compiler, POSIX, Unix

ACM Reference Format:
Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. 2021. PaSh: Light-touch Data-Parallel Shell Processing. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3447786.3456228

*The two marked authors contributed equally to the paper.

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456228

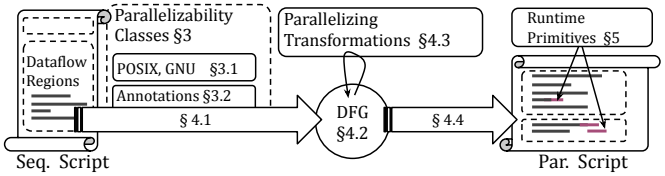


Fig. 1. PaSh overview. PaSh identifies dataflow regions (§4.1), converts them to dataflow graphs (§4.2), applies transformations (§4.3) based on the parallelizability properties of the commands in these regions (§3.1, §3.2), and emits a parallel script (§4.4) that uses custom primitives (§5).

1 Introduction

The Unix shell is an environment—often interactive—for composing programs written in a plethora of programming languages. This language-agnosticism, coupled with Unix's toolbox philosophy [33], 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:

- *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` [34, 42, 50].
- *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` [52], `ts` [22], `qsub` [14], SLURM [60]; 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, 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.

This paper presents a system called PaSh and outlined in Fig. 1 for parallelizing POSIX shell scripts that benefits both programmer groups, with emphasis on shell 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.

These components 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 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 on 44 unmodified scripts including (i) a series of smaller scripts, ranging from classic Unix one-liners to modern data-processing pipelines, and (ii) two large and complex use cases for temperature analysis and web indexing. 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), the latter traditionally falling 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 presenting an overview of 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), later used to explore 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 [38], 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 [59, §2.1]. To enable such a succinct program composition, Unix incorporates several features.

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.

```
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.

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, 16, 62], 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 program invokes `cut` with two separate sets of flags.

Scripts Another challenge is due to the language of the POSIX shell. First, the language contains constructs that enforce sequential execution: The sequential composition operator (`;`) in Fig. 2 indicates that the assignment to `base` must be 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 outputs a new POSIX script that incorporates data parallelism. The degree of data parallelism sought by PaSh is configurable using a `--width` parameter, whose default value is system-specific. Fig. 3 highlights a few fragments of the parallel script resulting from applying PaSh with `--width=2` to the script of Fig. 2—resulting in 2 copies of {`grep`, `tr`, `cut`, *etc.*}.

PaSh first identifies sections of the script that are potentially parallelizable, *i.e.*, 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.

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 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 *dataflow regions* containing commands that are candidates for parallelization (§4.1). This analysis is guided by the script structure: some constructs expose parallelism (*e.g.*, `&`, `|`); others enforce synchronization (*e.g.*, `;`, `||`). 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

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

```
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 the parallel execution through named pipes, parallel operators, and custom runtime primitives—*e.g.*, `eager`, `split`, and `get-pids`.

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 Coreutils (§3.1). Building on this study, it develops a lightweight extensibility framework that enables light-touch parallelization of a command 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 of 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).

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

Class	Key	Examples	Coreutils	POSIX
Stateless	Ⓢ	`tr`, `cat`, `grep`	13 (12.5%)	19 (12.7%)
Parallelizable Pure	Ⓟ	`sort`, `wc`, `head`	17 (16.3%)	13 (8.7%)
Non-parallelizable Pure	Ⓝ	`sha1sum`	13 (12.5%)	11 (7.3%)
Side-effectful	Ⓔ	`env`, `cp`, `whoami`	61 (58.6%)	105 (70.4%)

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 aligned 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 as 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 within a single data stream.[2] This is because their internal state depends on prior state in non-trivial ways over the same pass. 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 main reasons. First, it includes commands related to the file-system—a central abstraction of the Unix design and philosophy [46]. In fact, Unix uses the file-system as a proxy to several file-unrelated operations such as access control and device driving. Second, this class 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 Framework

To address the challenge of a language-agnostic environment (§2.2), PaSh allows communicating key details about their parallelizability through a lightweight extensibility framework comprising two components: an annotation language, and an interface for developing parallel command aggregators. The framework can be used both by developers of new commands as well as developers maintaining existing commands. The latter group can express additions or changes to the command's implementation or interface, which is important as commands are maintained or extended over long periods of time.

The extensibility framework is expected to be used by individuals who understand the commands and their parallelizability properties, and thus PaSh assumes their correctness. The framework could be used as a foundation for crowdsourcing the annotation effort, for testing annotation records, and for generating command aggregators. We use this extension framework in a separate work to synthesize command aggregators automatically [57].

[2] Note that these commands may still be parallelizable across different data streams, for example when applied to different input files.

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.

Manipulating a shell script in its original form to expose parallelism is challenging as each command has a different interface. Some commands read from standard input, while others read from input files. Ordering here is important, as a command may read several inputs in a predefined input order. For example, `grep "foo" f1 - f2` first reads from `f1`, then shifts to its standard input, and finally reads `f2`.

Additionally, commands expose flags or options for allowing users to control their execution. Such flags may directly affect a command's parallelizability classification as well as the order in which it reads its inputs. 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.

To address all these concerns, PaSh introduces an annotation language encoding first-order logic predicates. The language allows specifying the aforementioned information, *i.e.*, correspondence of arguments to inputs and outputs and the effects of flags. Annotations assign one of the four parallelizability class as a default class, subsequently refined by the set of flags the command exposes. Additionally, for commands in Ⓢ and Ⓟ, the language captures how a command's arguments, standard input, and standard output correspond to its inputs and outputs. Annotations in these classes can also express ordering information about these inputs—effectively lifting commands into a more convenient representation where they only communicate with their environment through a list of input and output files.

The complete annotation language currently contains 8 operators, one of which supports regular expressions. It was used to annotate 47 commands, totaling 708 lines of JSON—an effort that took about 3–4 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.

Example Annotations Two commands whose annotations sit at opposing ends of the complexity spectrum are `chmod` and `cut`. The fragment below shows the annotation for `chmod`.

```
{ "command": "chmod",
  "cases": [ { "predicate": "default",
               "class": "side-effectful" } ] }
```

This annotation is simple, but serves as an illustration of the annotation structure. Each annotation is a JSON record that contains the command name, and a sequence of cases. Each case contains a predicate that matches on the arguments of the command invocation. It assigns a parallelizability class (C1) to a specific command instance, *i.e.*, the combination of its inputs-output consumption (C2) and its invocation arguments (C3). In this case, `chmod` is side-effectful, and thus the `"default"` predicate of its single `cases` value always matches—indicating the presence of side-effects.

The annotation for `cut` is significantly more complex, and is only shown in part (the full annotation is in Appendix B). This annotation has two cases, each of which consists of a predicate on `cut`'s arguments and an assignment of its parallelizability class, inputs, and outputs as described above. We only show `cut`'s first predicate, slightly simplified for clarity.

```
{ "predicate": {"operator": "exists", "operands": [ "-z" ]},
  "class": "n-pure",
  "inputs": [ "args[:]" ],
  "outputs": [ "stdout" ] }
```

This predicate indicates that if `cut` is called with `-z` as an argument, then it is in Ⓝ, *i.e.*, it only interacts with the environment by writing to a file (its `stdout`) but cannot be parallelized. This is because `-z` forces `cut` to delimit lines with `NUL` instead of newline, meaning that we cannot parallelize it by splitting its input in the line boundaries. The case also indicates that `cut` reads its inputs from its non-option arguments.

Experienced readers will notice that `cut` reads its input from its `stdin` if no file argument is present. This is expressed in the `"options"` part of `cut`'s annotation, shown below:

```
{ "command": "cut",
  "cases": [ ... ],
  "options": [ "empty-args-stdin",
               "stdin-hyphen" ] }
```

Option `"empty-args-stdin"` indicates that if non-option arguments are empty, then the command reads from its `stdin`. Furthermore, option `"stdin-hyphen"` indicates that a non-option argument that is just a dash `-` represents the stdin.

The complete annotation in Appendix B) shows the rest of the cases (including the default case for `cut`, which indicates that it is in Ⓢ).

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.

Most commands only need an *aggregate* function, as the *map* function for many commands is the sequential command itself. PaSh defines a set of aggregators for many POSIX and GNU commands in Ⓟ. This set doubles as both PaSh's standard library and an exemplar for community efforts tackling other commands. Below is the Python code for one of the simplest *aggregate* functions, the one for `wc`:

```
#!/usr/bin/python
import sys, os, functools, utils

def parseLine(s):
  return map(int, s.split())

def emitLine(t):
  f = lambda e: str(e).rjust(utils.PAD_LEN, ' ')
  return [" ".join(map(f, t))]

def agg(a, b):
  # print(a, b)
  if not a:
    return b
  az = parseLine(a[0])
  bz = parseLine(b[0])
  return emitLine([ (i+j) for (i,j) in zip(az, bz) ])

utils.help()
res = functools.reduce(agg, utils.read_all(), [])
utils.out("".join(res))
```

The core of the aggregator, function `agg`, takes two input streams as its arguments. The `reduce` function lifts the aggregator to arity n to support an arbitrary number of parallel *map* commands. This lifting allows developers to think of aggregators in terms of two inputs, but generalize them to operate on many inputs. Utility functions such as `read` and `help`, common across PaSh's aggregator library, deal with error handling when reading multiple file descriptors, and offer a `-h` invocation flag that demonstrates the use of each aggregator.

PaSh's library currently contains over 20 aggregators, many of which are usable by more than one command or flag. For example, the aggregator shown above is shared among `wc`, `wc -lw`, `wc -lm`, *etc.*

4 Dataflow Graph Model

PaSh's core is an abstract dataflow graph (DFG) model (§4.2) used as the intermediate representation on which PaSh performs parallelism-exposing transformations. PaSh first lifts sections of the input script to the DFG representation (§4.1), then performs transformations to expose parallelism (up to the desired `--width`) (§4.3), and finally instantiates each DFG back to a parallel shell script (§4.4). A fundamental characteristic of PaSh's DFG is that it encodes the order in which a node reads its input streams (not just the order of input

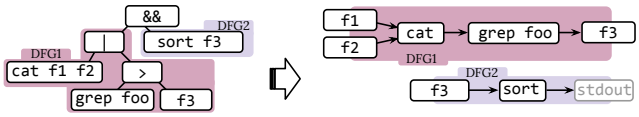


Fig. 4. From a script AST to DFGs. The AST on the left has two dataflow regions, each not extending beyond && (*Cf.*§4.1). Identifiers `f1`, `f2`, and `f3` sit at the boundary of the DFG.

elements per stream), which in turn enables a set of graph transformations that can be iteratively applied to expose parallelization opportunities for Ⓢ and Ⓟ commands.

To the extent possible, this section is kept informal and intuitive. The full formalization of the dataflow model, the shell↔DFG bidirectional translations, and the parallelizing transformations, as well as their proof of correctness with respect to the script's sequential output, are all presented in a separate work [20].

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 (Fig. 4):

```
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. Intuitively, dataflow regions correspond to sub-expressions of the program that would be allowed to execute independently by different processes in the POSIX standard [18]. 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 translation time. Examples of such information include the values of environment variables, unexpanded strings, and sub-shell constructs. For the sake of correctness, PaSh takes a conservative approach and avoids parallelizing nodes for which it has incomplete information. It will not attempt to parallelize sub-expressions for 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) data streams, *i.e.*, files or pipes, and (ii) commands, communicating through these streams.

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. For example, the edges in DFG1 of Figure 4 are the files `f1`, `f2`, and `f3`, as well as the unnamed pipe that connects `cat` and `grep`. We fix the data quantum to be character lines, *i.e.*, sequences of characters followed by the newline character,[3] so edges represent possibly unbounded sequences of lines. As seen above, an edge can either refer to a named file, an ephemeral pipe, or a Unix FIFO used for interprocess communication. Edges that do not start from a node in the graph represent the graph inputs; edges that do not point to a node in the graph represent its outputs.

Nodes—Commands A node of the graph represents a relation (to capture nondeterminism) from a possibly empty list of input streams to a list of output streams. This representation captures all the commands in the classes Ⓢ, Ⓟ, and Ⓝ, since they only interact with the environment by reading and writing to streams. We require that nodes are monotone, namely that they cannot retract output once they have produced it. As an example, `cat`, `grep`, and `sort` are the nodes in the DFGs of Figure 4.

Streaming Commands A large subset of the parallelizable Ⓢ and Ⓟ classes falls into the special category of streaming commands. These commands have two execution phases. First, they consume a (possibly empty) set of input streams that act as configuration. Then, they transition to the second phase where they consume the rest of their inputs sequentially, one element at a time, in the order dictated by the

[3] This is a choice that is not baked into PaSh's DFG model, which supports arbitrary data elements such as characters and words, but was made to simplify alignment with many Unix commands.

Fig. 5. Stateless parallelization transformation. The `cat` node is commuted with the stateless node to exploit available data parallelism.

configuration phase and produce a single output stream. The simplest example of a streaming command is `cat`, which has an empty first phase and then consumes its inputs in order, producing their concatenation as output. A more interesting example is `grep` invoked with `-f patterns.txt` as arguments; it first reads `patterns.txt` as its configuration input, identifying the patterns for which to search on its input, and then reads a line at a time from its standard input, stopping when it reaches EOF.

4.3 Graph Transformations

PaSh defines a set of semantics-preserving graph transformations that act as parallelism-exposing optimizations. Both the domain and range of these transformations is a graph in PaSh's DFG model; 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 lines of their input stream without maintaining any state (§3.1). To avoid referring to the internal command state, we can instead determine that a command is stateless if its output is the same if we "restart" it after it has read an arbitrary prefix of its input. If a command was stateful, then it would not produce the same output after the restart. Formally, a streaming command f is stateless if it commutes with the operation of concatenation on its streaming input, *i.e.*, it is a semigroup homomorphism:

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

In the above $x \cdot x'$ is the concatenation of the two parts of f's streaming input and c is the configuration input (which needs to be passed to both instances of f). The above equation 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. Note that we only focus on deterministic stateless commands and that is why f is a function and not a relation in the above.

Pure commands such as `sort` and `wc` can also be parallelized, using divide-and-conquer parallelism. These commands can be applied independently on different segments of their inputs, and then their outputs are aggregated to produce the final result. More formally, these pure commands f can be implemented as a combination of a function *map* and an associative function *aggregate* that satisfy the following equation:

$$\forall x, x', c, f(x \cdot x', c) = aggregate(map(x, c), map(x', c), c)$$

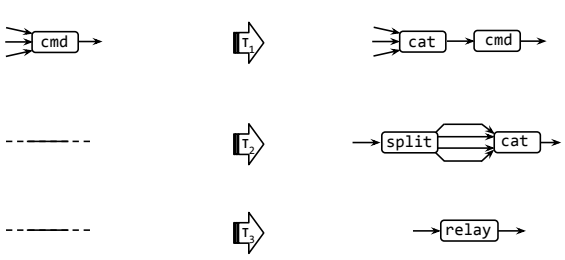


Fig. 6. Auxiliary transformations. These augment the DFG with `cat`, `split`, and `relay` nodes.

Parallelization Transformations Based on these equations, we can define a parallelization transformation on a node $f \in$ Ⓢ whose streaming input is a concatenation, *i.e.*, produced using the command `cat`, of n input streams and is followed by a node f' (Fig. 5). The transformation replaces f 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 f'. Formally:

$$v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow v(x_1, s) \cdot v(x_2, s) \cdots v(x_n, s)$$

The transformation can be extended straightforwardly to nodes $v \in$ Ⓟ, implemented by a $(map, aggregate)$ pair:

$$v(x_1 \cdot x_2 \cdots x_n, s) \Rightarrow \\ aggregate(map(x_1, s), map(x_2, s), \ldots map(x_n, s), s)$$

Both transformations can be shown to preserve the behavior of the original graph assuming that the pair $(map, aggregate)$ meets the three invariants outlined earlier (§3.2) and the aforementioned equations hold.

Auxiliary Transformations PaSh also performs a set of auxiliary transformations t_{1-3} that are depicted in Fig. 6. 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 performance improvements (§5), as well as for monitoring and debugging.

Degree of Parallelism The degree of parallelism achieved by PaSh is affected by the width of the final dataflow graph. The dataflow width corresponds, intuitively, to the number of data-parallel copies of each node of the sequential graph and thus the fanout of the `split` nodes that PaSh introduces. The dataflow width is configured using the `--width` parameter, which can be chosen by the user depending on their script characteristics, input data, and target execution environment. By default, PaSh assigns width to 2 if it is executing on a machine with 2-16 processors, and `floor(cpu_cores/8)` if it is executing on a machine with more than 16 processors.

This is a conservative limit that achieves benefits due to parallelism but does not consume all system resources. It is not meant to be optimal, and as shown in our evaluation, different scripts achieve optimal speedup with different `--width` values, which indicates an interesting direction for future work.

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 the 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
```

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. 7a).

To solve this, one might be tempted to replace FIFOs with files, a central Unix abstraction, simulating pipes of arbitrary buffering (Fig. 7b). Aside from severe performance implications, naive replacement can lead to subtle race conditions, as a consumer may reach EOF before a producer. Alternatively, consumers could wait for producers to complete before opening the file for reading (Fig. 7c); however, this would insert artificial barriers impeding task-based parallelism and wasting disk resources—that is, this approach allows for data parallelism to the detriment of task parallelism.

To address this challenge, PaSh inserts and instantiates eager `relay` nodes at these points (Fig. 7d). 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. 7.

Splitting Challenges To offer data parallelism, PaSh needs to split an input data stream to multiple chunks operated upon in parallel. Such splitting is needed at least once at the beginning of a parallel fragment, and possibly every time within the parallel program when an *aggregate* function of a stage merges data into a single stream.

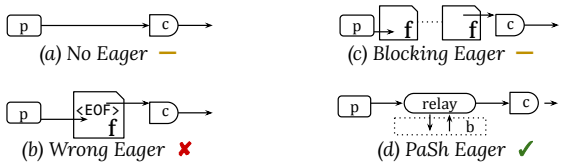


Fig. 7. Eager primitive. Addressing intermediary laziness is challenging: (a) FIFOs are blocking; (b) files alone introduce race conditions between producer/consumer; (c) files + `wait` inhibit task parallelism. Eager `relay` nodes (d) address the challenge while remaining within the PaSh model.

To achieve this, PaSh's transformations insert split nodes that correspond to a custom `split` command. For `split` to be effective, it needs to disperse its input uniformly. PaSh does not do this in a round-robin fashion, as that would require augmenting the data stream with additional metadata to maintain FIFO ordering—a challenge for both performance and correctness. PaSh instead splits chunks in-order, which necessitates knowledge of the input size beforehand and which is not always available. To address this challenge, PaSh provides a `split` implementation 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 for the last one) to address laziness as described above.

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 notification 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 handling is different from the error handling for other system calls and unusual compared to non-Unix systems[4] 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 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 the second `cat` 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.

This problem is not unique to PaSh; it occurs even when manually parallelizing scripts using FIFOs (but not when using *e.g.*, intermediary files, *Cf.* §5, Laziness). It is exacerbated, however, by PaSh's use of the `cat fifo1 fifo2` construct, used pervasively when parallelizing commands in Ⓢ.

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

Tab. 2. Summary of Unix 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	0.335s	4.560s	comparisons (`comm`)
difference	2 × Ⓢ, 2 × Ⓟ, Ⓝ	10 GB	25m49.097s	125	509	0.186s	2.341s	non-parallelizable `diffing`
bi-grams	3 × Ⓢ, 3 × Ⓟ	3 GB	38m9.922s	185	761	0.313s	4.310s	stream shifting and merging
set-difference	5 × Ⓢ, 2 × Ⓟ, Ⓝ	10 GB	51m32.313s	155	635	0.316s	4.358s	two pipelines merging to a `comm`
sort-sort	Ⓢ, 2 × Ⓟ	10 GB	31m26.147s	154	634	0.293s	3.255s	parallelizable Ⓟ after Ⓟ
shortest-scripts	5 × Ⓢ, 2 × Ⓟ	85 MB	28m45.900s	142	574	0.328s	4.657s	long Ⓢ pipeline ending with Ⓟ

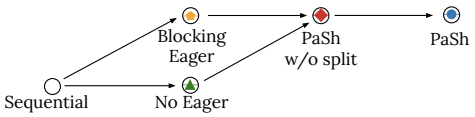


Fig. 8. Runtime setup lattice. Parallel No Eager and Blocking Eager improve over sequential, but are not directly comparable. PaSh w/o Split adds PaSh's optimized `eager` relay, and PaSh uses all primitives in §5 (Fig. 9).

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 As discussed earlier, 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 micro-benchmarks for targeted comparisons.

Highlights This paragraph highlights results for `width=16`, but PaSh's evaluation reports on varying `widths` (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 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.

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.

We note a few characteristics of our setup that minimize statistical non-determinism: (1) our evaluation experiments take several hours to complete (about 23 hours for the full set), (2) our experimental infrastructure is hosted on premises, not shared with other groups or researchers, (3) the runtime does not include any managed runtimes, virtualization, or containment,[5] (4) many commands are repeated many times—for example, there are more than 40 instances of `grep` in our benchmark set. The set of benchmarks also executes with smaller inputs multiple times a week (using

[5] While PaSh is available via Docker too, all results reported in this paper are from non-containerized executions.

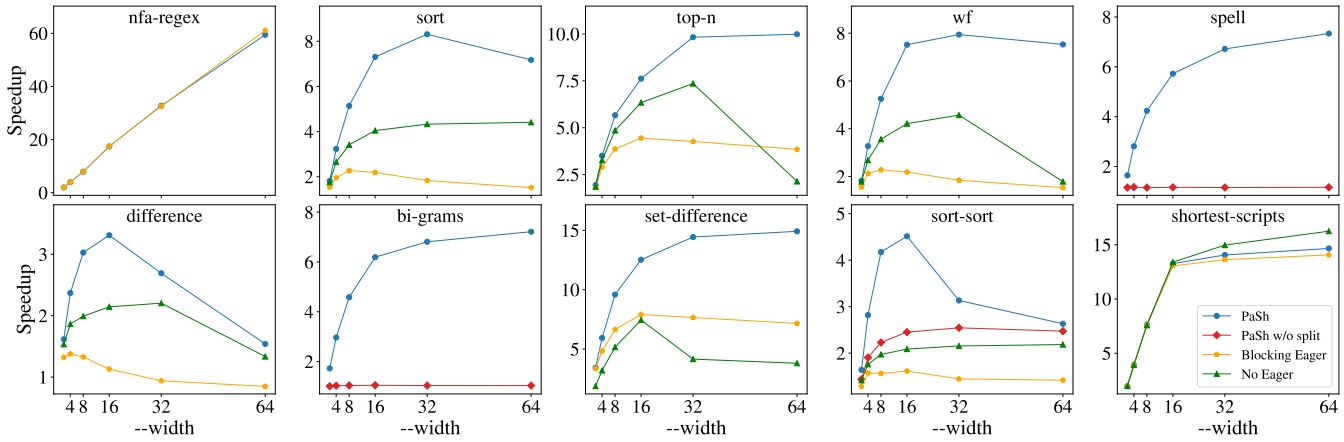


Fig. 9. PaSh's speedup for `width`=2–64. Different configurations per benchmark: (i) PaSh: the complete implementation with `eager` and `split` enabled, (ii) PaSh w/o split: `eager` enabled (no `split`), (iii) Blocking Eager: only blocking `eager` enabled (no `split`), (iv) No Eager: both `eager` and `split` disabled. For some pairs of configurations, PaSh produces identical parallel scripts and thus only one is shown.

continuous integration), reporting minimal statistical differences between runs.

Parallelism PaSh's degree of parallelism is configured by the `--width` flag (§4.3). 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 `width` smaller than the physical cores available in our 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, 53]. 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 all points on the runtime configuration lattice (Fig. 8).

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` [53, 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. 9 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 where PaSh does not add `split`, as the width of the DFG is constant; conversely, when both lines are shown (*e.g.*, Spell, Bi-grams, and Sort), PaSh has added `splits` 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, beyond which PaSh w/o Split configurations are sequential and thus see no speedup. Finally, Tab. 2 shows that PaSh's transformation time is negligible, and its COST [35], *i.e.*, the degree of parallelism threshold over which PaSh starts providing absolute execution time benefits, 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. PaSh's `eager` primitive improves over No 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) PaSh w/o Split offers benefits despite the lack of `split` because it fully parallelizes the first `sort`, and (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 *aggregate*, and 14 `eager` processes.

Take-aways PaSh accelerates scripts by up to 60×, depending on the characteristics of the commands involved in a script. Its runtime constructs improve over the baseline speedup achieved by its parallelization 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 [29] solvable by Unix pipelines. The problems were designed to highlight Unix's modular philosophy [33]. 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. 10 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 (#0–30) in Fig. 10. 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 targeted expressiveness of the replacement commands can be exploited by PaSh—in this specific case, achieving 8.1× speedup (*vs.* the original 1.01×).

For all other scripts (#0–24), PaSh's speedup is capped due to a combination of reasons: (i) scripts 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) scripts are deep pipelines that already exploit task parallelism (#4, 10, 11, 13, 15, 17, 19, 21, 22), or (iii) scripts 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 performance. 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).

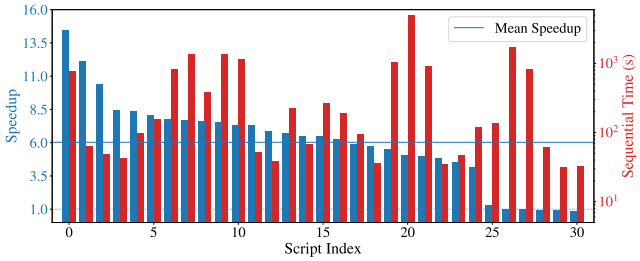


Fig. 10. Unix50 scripts. Speedup (left axis) over sequential execution (right axis) for Unix50 scripts. Parallelism is 16× on 10GB of input data (*Cf.*§6.2). Pipelines are sorted in descending speedup order.

Program This program is inspired by the central example in "Hadoop: The Definitive Guide" [59, §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× speedup (vs. 33m58s) for all the pre-processing (75% of the total running time) and 12.31× speedup (vs. 10m4s) for computing the maximum.

Discussion The speedup of the preprocessing phase of the pipeline is bound by the network and I/O costs since `curl` downloads 82GB of data. However, the speedup for the processing phase (CPU-bound) is 12.31×, much higher than what would be achieved by parallelizing per year (for a total of five years). 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. 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.

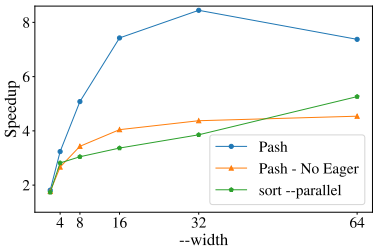
Discussion The original script contains 34 pipeline stages, thus the sequential version already benefits from task-based

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 First, we compare a `sort` parallelized by PaSh (S_p) against the same `sort` invoked using the `--parallel` flag set (S_g).[6] While the `--parallel` flag 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 [52], 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.

[6] Both `sorts` use the same buffer size internally [44].

7 Related Work

Existing techniques for exploiting parallelism are not directly comparable to PaSh, because they either 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` [14], SLURM [60], `parallel` [52]—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, 32, 49] 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, Greenberg [17] argued that the shell and its constructs can be seen as a DSL for orchestrating concurrent processes. PaSh's extraction of dataflow regions is based on a similar observation, but its central focus is on achieving data parallelism from these dataflow regions automatically.

Developed concurrently with PaSh, the Process-Offload SHell (POSH) [45] is a shell and runtime that automatically reduces data movement when running shell pipelines on data stored in remote storage á la NFS. POSH accelerates I/O-heavy pipelines that access files in remote filesystems, by offloading computation to servers closer to the data. PaSh is a shell-to-shell compiler that parallelizes Unix shell scripts running on a single multi-processor machine by transforming them to DFGs, applying transformations, and then transforming them back to parallel shell scripts augmented with PaSh's runtime primitives that are executed on the user's shell. Both PaSh and POSH observe that Unix commands can have arbitrary behaviors (§2.2), thus each introducing an annotation language that fits its problem: POSH uses annotations to identify which files are accessed by a pipeline, and thus co-locates commands and their dependencies; PaSh uses annotations to identify whether a command is parallelizable and, if so, how to translate it to a dataflow node. Both systems descend from a lineage of annotation-based black-box transformations [41, 54, 55, 61].

Low-level Parallelization There exists significant work on automating parallelization at the instruction level, starting with explicit `DOALL` and `DOACROSS` annotations [7, 30] and continuing with compilers that attempt to automatically extract parallelism [19, 40]. These efforts 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 [13, 15, 28] and interactive parallelization tools [24, 26]. 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, 62] and stream-processing [8, 37]. Systems implementing DFGs often perform optimizations that are correct given subtle assumptions on the dataflow nodes that do not always hold, introducing erroneous behaviors. Recent work [21, 25, 31, 47] attempts to address this issue by performing optimizations only in cases where correctness is preserved, or by testing that applied optimizations preserve the original behavior. PaSh draws inspiration from these efforts, in that it delegates the satisfaction of assumptions to the annotation writers, who are expected to be command developers rather than shell users (§1), ensuring that transformations preserve the behavior of the original dataflow. Its DFG model, however, is different from earlier efforts in that it explicitly captures and manipulates ordering constraints. The constraints are due to the intricacies of the Unix model—*e.g.*, FIFO streams, argument processing, and concatenation operators.

Parallel Userspace Environments By focusing on simplifying the development of distributed programs, a plethora of environments additionally assist in the construction of parallel software. Such systems [1, 36, 39], languages [27, 48, 58], or system-language hybrids [11, 43, 56] 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, 12, 16, 51] offer fully automated parallelism as long as special primitives are used—*e.g.*, map-reduce-style primitives for Phoenix [51]. These primitives make strong assumptions about the nature of the computation—*e.g.*, commutative and associative aggregation functions that can be applied on their inputs in any order. By targeting specific classes of computation (*viz.* PaSh's parallelizability), these primitives are significantly optimized for their target domains. PaSh instead chooses an approach that is better tailored to the shell: it does not require rewriting parts of a shell script using specific parallelization-friendly primitives, but rather lifts arbitrary commands to a parallelization-friendly space using an annotation framework.

Dryad [23] 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, use blocks written in 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, Nick Roessler, Sage Gerard, and several open-source contributors. We are grateful to our shepherd, Julia Lawall, for her guidance. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF awards CCF 1763514 and 2008096. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA or NSF.

References

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

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

[3] Jon Bentley. 1985. Programming Pearls: A Spelling Checker. *Commun. ACM* 28, 5 (May 1985), 456–462. https://doi.org/10.1145/3532.315102

[4] Jon Bentley, Don Knuth, and Doug McIlroy. 1986. Programming Pearls: A Literate Program. *Commun. ACM* 29, 6 (June 1986), 471–483. https://doi.org/10.1145/5948.315654

[5] Pawan Bhandari. 2020. Solutions to unixgame.io. https://git.io/Jf2dn Accessed: 2020-04-14.

[6] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.* 23, 3 (2004), 777–786. https://doi.org/10.1145/1015706.1015800

[7] Michael Burke and Ron Cytron. 1986. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN '86)*. ACM, New York, NY, USA, 162–175. https://doi.org/10.1145/12276.13328

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[10] Tom Duff. 1990. Rc-A shell for Plan 9 and Unix systems. *AUUGN* 12, 1 (1990), 75.

[11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 118–129. https://doi.org/10.1145/2034675.2034690

[12] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR* 8 (2009).

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

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

[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. 2002. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, Vol. 36. ACM, 291–303.

[16] 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. 2002. 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)*. Association for Computing Machinery, New York, NY, USA, 291âĂŞ303. https://doi.org/10.1145/605397.605428

[17] Michael Greenberg. 2018. The POSIX shell is an interactive DSL for concurrency. https://cs.pomona.edu/~michael/papers/dsldi2018.pdf.

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

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

[20] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin Rinard. 2020. An Order-aware Dataflow Model for Extracting Shell Script Parallelism. *arXiv preprint arXiv:2012.15422* (2020).

[21] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys (CSUR)* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

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

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

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

[25] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: Differential Output Testing for Stream Processing Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

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

[27] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/1250734.1250755

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

[29] Nokia Bell Labs. 2019. The Unix Game—Solve puzzles using Unix pipes. https://unixgame.io/unix50 Accessed: 2020-03-05.

[30] Amy W. Lim and Monica S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 201–214. https://doi.org/10.1145/263699.263719

[31] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. 2019. Data-Trace Types for Distributed Stream Processing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 670–685. https://doi.org/10.1145/3314221.3314580

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

[33] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. 1978. UNIX Time-Sharing System: Foreword. *Bell System Technical Journal* 57, 6 (1978), 1899–1904.

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

[35] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[36] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53. https://www.cs.cornell.edu/home/rvr/papers/Amoeba1990s.pdf

[37] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[38] National Oceanic and Atmospheric Administration. 2017. National Climatic Data Center. https://www.ncdc.noaa.gov/.

[39] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite network operating system. *Computer* 21, 2 (1988), 23–36. http://www.research.ibm.com/

people/f/fdouglis/papers/sprite.pdf

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

[41] Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 291–305. https://doi.org/10.1145/3341301.3359652

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

[43] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. 1990. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*. 1–9. http://css.csail.mit.edu/6.824/2014/papers/plan9.pdf

[44] Pixelbeat. 2015. Answer to: Sort –parallel isn't parallelizing. https://superuser.com/a/938634 Accessed: 2020-04-14.

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

[46] Dennis M. Ritchie and Ken Thompson. 1973. The UNIX Time-sharing System. *SIGOPS Oper. Syst. Rev.* 7, 4 (Jan. 1973), 27–. https://doi.org/10.1145/957195.808045

[47] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 504–517. https://doi.org/10.1109/TC.2013.221

[48] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1086365.1086370

[49] Diomidis Spinellis and Marios Fragkoulis. 2017. Extending Unix Pipelines to DAGs. *IEEE Trans. Comput.* 66, 9 (2017), 1547–1561.

[50] Richard M Stallman and Roland McGrath. 1991. GNU Make—A Program for Directing Recompilation. https://www.gnu.org/software/make/manual/make.pdf.

[51] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. 2011. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications (MapReduce '11)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/1996092.1996095

[52] Ole Tange. 2011. GNU Parallel—The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. https://doi.org/10.5281/zenodo.16303

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

[54] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. 2019. Ignis: Scaling Distribution-oblivious Systems with Light-touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1010–1026. https://doi.org/10.1145/3314221.3314586

[55] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131

[56] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. 2015. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=2831090.2831105

[57] Nikos Vasilakis, Jiasi Shen, and Martin Rinard. 2020. Automatic Synthesis of Parallel and Distributed Unix Commands with KumQuat. *arXiv preprint arXiv:2012.15443* (2020).

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

[59] Tom White. 2015. *Hadoop: The Definitive Guide* (4th ed.). O'Reilly Media, Inc.

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

[61] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 293–306. https://www.usenix.org/conference/atc20/presentation/yuan

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=2228298.2228301

A Annotation for the Command cut

The code below shows the full annotation for `cut`.

```
{ "command": "cut",
  "cases": [
    { "predicate": {
        "operator": "or",
        "operands": [
          { "operator": "val_opt_eq",
            "operands": [ "-d", "\n" ] },
          { "operator": "exists",
            "operands": [ "-z" ] }
        ]
      },
      "class": "pure",
      "inputs": [ "args[:]" ],
      "outputs": [ "stdout" ]
    },
    { "predicate": "default",
      "class": "stateless",
      "inputs": [ "args[:]" ],
      "outputs": [ "stdout" ]
    }
  ],
  "options": [ "stdin-hyphen", "empty-args-stdin" ],
  "short-long": [
    { "short": "-d", "long": "--delimiter" },
    { "short": "-z", "long": "--zero-terminated" }
  ]
}
```

B Artifact Appendix

Summary The artifact consists of several parts: (i) a mirror of PaSh' GitHub repository (git commit `e5f56ec`, available permanently in branch `eurosys-2021-aec-frozen`) including annotations, the parallelizing compiler, and the runtime primitives presented in this paper; (ii) instructions for pulling

Tab. 3. Major experiments presented in the paper. There are four major experiments presented in the paper: (i) Common Unix one-liners, (ii) Unix50 from Bell Labs, (iii) NOAA Weather Analysis, and (iv) Wikipedia Web Indexing.

Experiment	Section	Location
Common Unix one-liners	§6.1	https://git.io/JYi9m
Unix50 from Bell Labs	§6.2	https://git.io/JYi9n
NOAA Weather Analysis	§6.3	https://git.io/JYi9C
Wikipedia Web Indexing	§6.4	https://git.io/JYi98

code and experiments, building from source, preparing the environment, and running the experiments; (iii) a 20-minute video walk-through of the entire artifact; and (iv) instructions for directly pulling a pre-built Docker container and building a Docker image from scratch; (v) scripts, descriptions, and instructions to run the experiments (automatically or manually) to reproduce the graphs and results presented in the paper.

Codebase information Below is a summary of key information about PaSh's repository:

- Repository: https://github.com/andromeda/pash
- License: MIT
- Stats: 2,278 commits, from 14 contributors

Artifact requirements Below is a summary of requirements for running PaSh and its evaluation experiments:

- CPU: a modern multi-processor, to show performance results (the more cpus, the merrier)
- Disk: about 10GB for small-input (quick) evaluation, about 100GB+ for full evaluation
- Software: Python 3.5+, Ocaml 4.05.0, Bash 5+, and GNU Coreutils (details below)
- Time: about 30min for small-input, about 24h for full evaluation

Dependencies The artifact depends on several packages; on Ubuntu 18.04: libtool, m4, automake, opam, pkg-config, libffi-dev, python3, python3-pip, wamerican-insane, bc, bsdmainutils, curl, and wget. PaSh and its experimental and plotting infrastructure make use of the following Python packages: jsonpickle, PyYAML, numpy, matplotlib. Experiments and workloads have their own dependencies—*e.g.*, pandoc-2.2.1, nodejs, and npm (Web indexing), or p7zip-full (Wikipedia dataset).

Access PaSh is available via several means, including:

- Git: `git clone git@github.com:andromeda/pash.git`
- Docker: `curl img.pash.ndr.md | docker load`
- HTTP: `wget pkg.pash.ndr.md`
- Shell: `curl -s up.pash.ndr.md | sh`

Code Structure This repo hosts the core PaSh development. The artifact's directory structure is as follows:

- annotations: Parallelizability study and associated command annotations.
- compiler: Shell-dataflow translations and associated parallelization transformations.
- docs: Design documents, tutorials, installation instructions, *etc.*
- evaluation: Shell pipelines and example scripts used in the evaluation of PaSh.
- runtime: Runtime component—*e.g.*, eager, split, and associated aggregators.
- scripts: Scripts related to installation, continuous integration, deployment, and testing.

Calling PaSh To parallelize a script `hello-world.sh` with a parallelization degree of 2, from the top-level directory of the repository run:

```
./pa.sh hello-world.sh
```

PaSh will compile and execute `hello-world.sh` on the fly.

Tutorial To go through a longer tutorial, see docs/tutorial.

Available subcommands Run `./pa.sh --help` to get more information about the available PaSh subcommands:

```
Usage: pa.sh [-h] [--preprocess_only] [--output_preprocessed]
             [-c COMMAND] [-w WIDTH] [--no_optimize]
             [--dry_run_compiler] [--assert_compiler_success]
             [-t] [-p] [-d DEBUG] [--log_file LOG_FILE]
             [--no_eager] [--speculation {no_spec,quick_abort}]
             [--termination {clean_up_graph,drain_stream}]
             [--config_path CONFIG_PATH] [-v] [input]

Positional arguments:
 input                 The script to be compiled and executed.

optional arguments:
 -h, --help
                       Show this help message and exit.
 --preprocess_only
                       Pre-process (not execute) input script.
 --output_preprocessed
                       Output the preprocessed script.
 -c COMMAND, --command COMMAND
                       Evaluate the following COMMAND as a
                       script, rather than a file.
 -w WIDTH, --width WIDTH
                       Set degree of data-parallelism.
 --no_optimize
                       Not apply transformations over the DFG.
 --dry_run_compiler
                       Not execute the compiled script, even
                       if the compiler succeeded.
 --assert_compiler_success
                       Assert that the compiler succeeded
                       (used to make tests more robust).
 -t, --output_time
                       Output the time it took for every step.
 -p, --output_optimized
```

```
                      Output the parallel script for inspection.
-d DEBUG, --debug DEBUG
                      Configure debug level; defaults to 0.
--log_file LOG_FILE
                      Location of log file; defaults to stderr.
--no_eager
                      Disable eager nodes before merging nodes.
--termination {clean_up_graph,drain_stream}
                      Determine the termination behavior of the
                      DFG. Defaults to cleanup after the last
                      process dies, but can drain all streams
                      until depletion.
--config_path CONFIG_PATH
                      Determine the config file path, by
                      default 'PASH_TOP/compiler/config.yaml'.
-v, --version         Show program's version number and exit
```