Parallel and Distributed Just-in-Time Shell Script Compilation

by

Tammam Mustafa

SB, Electrical Engineering and Computer Science, Massachusetts Institute of Technology(2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Nikos Vasilakis
Research Scientist
Thesis Supervisor

Certified by .....
Martin C. Rinard
Professor of Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Parallel and Distributed Just-in-Time Shell Script Compilation

by

Tammam Mustafa

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In the past several years, the shell has received renewed interest from the research community. This thesis describes the work I did to advance the performance and capabilities of the current state-of-the-art shell-script parallelization systems. In the first half of this thesis, I focus on my contributions to PaSh-JIT, a JIT compiler for parallelizing POSIX shell scripts. In the second half, I explore the design and implementation of Distributed-PaSh, a shell that can utilize distributed computing resources and easily interface with distributed storage systems to efficiently execute data-processing pipelines. Distributed-PaSh analyzes the dataflow graph of a given script to create highly parallel data pipelines and execute those pipelines in a distributed cluster while giving special attention to data locality and movement. Distributed-PaSh achieves higher performance than single machine sequential and parallel shells.

Thesis Supervisor: Nikos Vasilakis
Title: Research Scientist

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science

Parallel and Distributed Just-in-Time Shell Script Compilation

by

Tammam Mustafa

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In the past several years, the shell has received renewed interest from the research community. This thesis describes the work I did to advance the performance and capabilities of the current state-of-the-art shell-script parallelization systems. In the first half of this thesis, I focus on my contributions to PaSh-JIT, a JIT compiler for parallelizing POSIX shell scripts. In the second half, I explore the design and implementation of Distributed-PaSh, a shell that can utilize distributed computing resources and easily interface with distributed storage systems to efficiently execute data-processing pipelines. Distributed-PaSh analyzes the dataflow graph of a given script to create highly parallel data pipelines and execute those pipelines in a distributed cluster while giving special attention to data locality and movement. Distributed-PaSh achieves higher performance than single machine sequential and parallel shells.

Thesis Supervisor: Nikos Vasilakis
Title: Research Scientist

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science

Acknowledgments

I would like thank my advisors, Prof. Martin Rinard and Nikos Vasilakis, for their supervision and support. Nikos introduced me to the world of research and taught me how to navigate ambiguity and to keep an eye on the bigger picture. I would also like to thank Konstantinos Kallas for being a great mentor and collaborator in the last two years, and for teaching me how to critically assess and evaluate research ideas. Finally, I would thank my family and friends for their support and encouragement; without them I wouldn't be here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Unix shell is a unique and fundamental environment. The shell allows programmers to compose scripts written in different languages and provides an easy and simple way of stringing a series of commands to create data pipelines that can utilize a single or multiple processors. This makes the shell the primary choice for specifying succinct and simple scripts for data processing, system orchestration, and other automation tasks.

Recent systems [31, 39, 33] accelerate such tasks by exploiting data parallelism: using *ahead-of-time* (AOT) analysis and transformation, these systems parse, analyze, and transform shell scripts into new scripts that execute in parallel. Unfortunately, AOT parallelization quickly becomes intractable due to the dynamic nature of the shell: dynamic features such as variable expansion and command substitution, pervasive in shell scripts, generate and consume values at run-time while depending on and interacting with the broader environment—*i.e.*, the filesystem, the environment variables, and the shell interpreter itself. Additionally, modern shells offer several different configurations and execution modes, leading to complex behaviors described in hundreds of pages of POSIX standards [2]. The complexity of these interactions and their side-effects lead existing parallelization tools to an unavoidable trade-off between (1) being conservative, aborting on scripts that use dynamic features, or (2) being

unsound, possibly breaking scripts during parallelization. Recent systems [31, 39, 33] tend to be conservative, operating only on fully expanded shell pipelines and having a hard time even on simple uses of variables (see §3.2). In first part of this thesis, I present PaSh-JIT [20], the first JIT compiler for the shell, focusing on my contributions around commutativity awareness and dependency untangling. PaSh-JIT achieves speed-ups of up to 33.7x over Bash and improves over PaSh-AOT by 2x on average.

Additionally, scaling out shell computations remains challenging: Whereas most developers can quickly stitch together shell scripts to compute on a single computer, scaling them out to multiple computers requires expert labor. As a result, shell scripts forego all of the benefits of distributed computing—the ability to speed up expensive computations or process data that would not fit into any single machine.

To understand this second problem, consider a shell script that calculates term frequencies over a set of inputs:

```
1 cat * | tr -cs A-Za -z\n | tr A-Z a-z | sort | uniq -c | sort -rn |
  ↪ head 5 > out
```

This program creates a character stream, breaks it into words, transliterates to lower-case, sorts to group duplicates, reduces duplicates to a counter, sorts counters in descending order, picks the top five results, and writes them to a file `out`. All these commands and other commands exist in the environment and can be written in any programming language. On a single computer and with a few GBs of input data, this pipeline takes a few seconds to execute; but scaling to multiple computers, this pipeline would need to be rewritten in one of the relevant models of distributed computation such as Hadoop [1] and Spark [42]. Only a few of these frameworks support language-agnostic third-party components, central among them Hadoop streaming [40]. Such manual rewriting is expensive and can introduce new bugs, cascading changes, and divergence from legacy functionality. In the second half of my thesis, I present Distributed-PaSh, the first JIT compiler for the shell capable of distributing shell scripts computations. I evaluate Distributed-PaSh against BaSh and

PaSh-JIT using 10 different scripts. I adapt the scripts to read from a distributed file system instead of local disk. Distributed-PaSh achieves a speedup of up to 8.17x over Bash and 4.65x over PaSh-JIT.

1.2 Contributions

This thesis makes two key contributions. First, it contributes commutativity awareness and dependency untangling to the PaSh-JIT system, both of which are critical to its performance. Second, it contributes a design and implementation of Distributed-PaSh, a JIT compiler for the shell that the can utilize cluster computing and distributed file systems to scale out shell scripts.

1.3 Thesis Outline

The thesis is organized as follows:

- Chapter 2 gives a brief background overview of the shell and related systems.
- Chapter 3 describes PaSh-JIT and my main contributions to the system.
- Chapter 4 explores the Distributed-PaSh design, implementation, and evaluation.
- Chapter 5 concludes my thesis and provides directions for future work.

Chapter 2

Background and Related Work

2.1 Shell Interface

The Unix shell is one of the most fundamental parts of any Unix OS. The shell language agnostic model and toolbox philosophy makes it one of the most dynamic environments to create and execute automation and data processing scripts. Shell scripts communicate using a variety of the following mechanisms, central among which:

1. Files using operator ($>$ File, $<$ File).
2. Named pipes (mkfifo).
3. Anonymous pipes (|).

These mechanisms, especially anonymous pipes, enable easy and efficient composition of data pipelines. A series of commands can be simply stringed together using the pipe operator to create complex data processing pipelines.

2.2 PaSh-AOT

PaSh-AOT uses semantic annotations to convert a script into a dataflow graph. The annotation conveys the behaviour of a command and their flags. The following classes are defined in the PaSh-AOT annotation system, ordered from easiest to parallelize to impossible:

- Stateless: commands that don't store any state across invocations. Examples: grep, cat, tr.

- Parallelizable Pure: commands that are purely functional but maintain an internal state. This class of commands could be parallelized given an aggregator function. Examples: sort, wc, head.

- Non-parallelizable Pure: commands that are purely functional but their state depends on prior state which makes parallelizing them non trivial. Examples: sha1sm.

- Side-effectful: commands that have side effects on the environment and system. Examples: cp, rm, env.

Flags can change a command's class. PaSh-AOT provides a domain-specific language to easily add annotations for commands and their flags. PaSh-AOT also provides annotations for the most common Unix commands and their flags. However, developers might still need to add annotations for less known commands or user-defined functions and executables.

This powerful, yet simple, annotation system allows PaSh-AOT to decide which parts of the dataflow graph can be split into multiple nodes that could run in parallel and at what point the output should be merged back to preserve correctness. PaSh-AOT optimizes the dataflow graph to push the merge node as far as possible to avoid the overhead of splitting and merging repeatedly. The system shows significant speedups (0.89–61.1×, avg: 6.7×) [39].

2.3 POSH

POSH focuses to shell applications with I/O-heavy components, particularly data processing workloads that read files from remote storage systems where networking and I/O are a major bottleneck. POSH offloads workloads to Proxy servers that are closest to the data. It also uses an annotation system for declaring splittable

commands across input arguments. This differs from PaSh-AOT in multiple ways, and perhaps easiest to show the difference between the two systems with an example:

- Consider the pipeline: `cat A | grep ‘‘the’’ | wc`. PaSh-AOT would split the file A into multiple chunks, each chunk would run in parallel through grep and wc. Finally a wc aggregator would sum up the results of the wc commands run in parallel and return the output. On the other hand POSH would run this full pipeline on a Proxy server closest to the machine where file A is stored.
- Now let's consider the pipeline: `cat A B C | grep ‘‘the’’ | wc`. Here POSH will execute the grep command on files A, B, C in parallel while utilizing the closest proxy server for each file to process that file. Finally, files will be merged and the wc command will run on the full output.

POSH focuses on improving performance by being aware of the data locations, utilizing proxy servers, and sometimes splitting the processing across multiple full files. While PaSh-AOT is mainly concerned with creating highly parallel data pipelines on the host machine even for a single file.

2.4 Parallel shell scripting

Recent work addresses significant challenges related to automatic shell script parallelization. POSH [31] and PaSh-AOT [39] are mostly-automated ahead-of-time shell-script parallelization systems; as described earlier, these systems focus on fully expanded shell pipelines that do not make use of dynamic features.

Recent work explored an order-aware dataflow model as a foundation for modeling the transformations these systems perform and proving them correct [16]. To enable divide-and-conquer parallelism, KumQuat [33] proposes a program-synthesis technique for generating aggregators for black-box commands.

PaSh-JIT builds on all this prior work, addressing fundamental limitations in static, ahead-of-time parallelization: AOT approaches apply to a very small subset of real shell scripts.

By opting for just-in-time parallelization, PaSh-JIT achieves parallel script behavior that is practically indistinguishable from the sequential execution—and ample opportunities for additional acceleration.

Other work on shell script parallelization either requires manual effort or is applicable to a smaller subset of scripts than our work. Such work includes: utilities like `qsub` [12], SLURM [41], and `parallel` [35]; shells with non-linear pipe topologies [11, 24, 34]; and using the shell itself as a DSL for concurrency [14].

2.5 Unix-related parallelization

There has been a significant body of work on parallel (and distributed) Unix and Unix-like environments [28, 26, 4], including shell-oriented efforts such as Plan9's `rc` [29]. Contrary to PaSh-JIT, these systems did not (aim to) offer full compatibility with the sequential Unix shell. They also focused on systems-level and program-runtime support, rather than automated program analyses and transformations.

2.6 Just-in-time compilation

Just-in-time compilation has been studied for long time [3], mainly in two contexts: (1) as a compilation technique for interpreted languages such as JavaScript [13], where critical type information is unavailable prior to execution; and (2) as a performance optimization over ahead-of-time compilation, allowing for specialization [37, 17], loop unrolling and function inlining [8, 30], and other profile-guided optimizations [27, 21]. PaSh-JIT draws inspiration from work in both contexts—resolving unavailable dynamic information at run-time and performing additional optimizations. It also leverages the optimistic compilation technique employed commonly by just-in-time compilers: when it fails to compile (parallelize), it simply runs the original fragment using the shell interpreter as a fallback option. PaSh-JIT differs from most JITs, dealing with different challenges: it operates at a higher level of abstraction, in a unique programming environment with no single unified runtime.

PaSh-JIT also draws inspiration from staged compilation [9] and partial evaluation [18]. These techniques perform some compilation ahead-of-time, waiting for the runtime to specialize and further optimize when there is more information about the environment of the target program and how it is used.

Chapter 3

PaSh-JIT

3.1 Introduction

The ahead-of-time compiler in PaSh-AOT works well for simple scripts, but its ability to parallelize a script becomes limited as the complexity of the scripts increases. This mainly stems from the dynamic nature of the shell; dynamic features in the shell, such as variable expansions and command substitution, are very common. The ahead-of-time compiler, in many cases, can't possibly know what a variable would refer to at later point in the script; this forces the compiler to make very conservative decisions in the initial compilation stage. PaSh-JIT addresses these limitations by creating a just-in-time compiler for the shell. PaSh-JIT achieves impressive speed-ups of up to 33.7× over Bash on a 64-core machine and improves the state of the art [39] (PaSh-AOT) by 2× on average.

This chapter gives an overview of the PaSh-JIT system and its main components as published in [20]. In the remaining parts, I focus on my contributions to it, namely, its commutativity awareness (§3.4) and dependency untangling (§3.5) capabilities.

3.2 Example & Overview

Below is a shell program that downloads a compressed archive of text files (books from Project Gutenberg), extracts them in a directory, and then performs an analysis

to find the frequencies of all words of a specific form.

```
IN=${IN:-$TOP/pg}
mkdir $IN
cd $IN
echo 'Download will take some time, be patient...'
wget $SOURCE/data/pg.tar.xz
if [ $? -ne 0 ]; then
  echo "Download failed!"
  exit 1
fi
cat pg.tar.xz | tar -xJ

cd $TOP
OUT=${OUT:-$TOP/output}
mkdir -p "$OUT"
for input in $(ls ${IN}); do
  cat "$IN/$input" | tr -sc '[A-Z][a-z]' '[\012*]' |
    grep '^....$' | sort | uniq -c > "${OUT}/${input}.out"
done
```

The program makes pervasive use of the shell's dynamic features. For example, it uses environment variables such as `$TOP`, variable expansion like `${OUT:-\$TOP/output}` to assign default values, command substitution `$(...)` as part of the loop condition, and state reflection on the file system by running `ls` on `$IN` (itself resolved dynamically).

None of the values of these variables can be known ahead of time just by analyzing the program's source code. They become known only at run-time, when the shell interpreter reaches these points in the program's execution. A sound AOT compiler such as PaSh-AOT [39] or POSH [31] would fail to parallelize—foregoing all the performance benefits of data-parallel execution spread across many files in `$IN`.

PaSh-JIT instead takes a JIT approach that interjects parallelization opportunities during and throughout the script's execution (Fig. 3-1).

3.3 Design

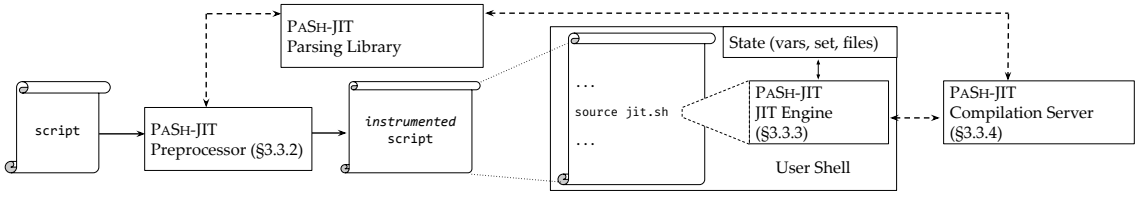

Figure 3-1: PaSh-JIT architecture overview

PaSh-JIT addresses the limitation of PaSh-AOT by moving the compilation from ahead-of-time to just-in-time. The JIT compiler aims to make parallelization decisions at a later point during the script execution; this allows the compiler to safely expand and know the values of variables and other substitutions and thus can make better decisions. If at a later point, a variable changes its value, the compiler is able to make accurate and less conservative decisions based on the new assignment.

3.3.1 Component overview

The following are the PaSh-JIT components:

Parsing Library: the parsing library contains Python bindings for the dash parser [15] which translates dash's AST to a Python AST and an implementation unparsing to translate to shell script.

Preprocessor: the preprocessor parses the shell script and instruments it with calls to the parallelizing compilation server.

Annotations: The annotations used by the compiler depends on PaSh-AOT annotations for dataflow graph parallelization.

JIT Engine: The JIT engine works through the interaction of the instrumented calls and the stateful long-lived compilation server.

The Compilation Server: A stateful long lived server that handles compilation requests for parallelizing regions of the script.

3.3.2 Preprocessor

Dynamic script interposition without any shell-interpreter modifications is hard.

To achieve this, PaSh-JIT opts for a light-weight script instrumentation preprocessing step: it marks *possible* parallelizable regions with code that dynamically determines whether or not to invoke the compiler.

The intuition behind PaSh-JIT's preprocessor is that a syntactic analysis of a shell script is enough to suggest potential parallelizable regions. This analysis is imprecise: there is no way to determine whether a command invocation will be pure ahead of time. Its goal however, is not to find parallelizable regions exactly, but rather to find potential compilation sites—PaSh-JIT sorts out the details at run-time, using up-to-date information about the system's state.

There is a trade-off when choosing the right size for these regions: the larger the region, the more opportunities exist for analysis and optimization but the less likely it is for the entire region to be parallelizable. PaSh-JIT targets a middle-ground: maximal schedule-free regions—*i.e.*, command sequences composed using shell primitives that do not impose scheduling restrictions. By focusing on maximal schedule-free regions, PaSh-JIT minimizes the number of compiler invocations and maximizes the cross-command parallelization opportunities for the compiler.

The preprocessor finds maximal possible parallelizable regions by searching the AST bottom-up, combining subtrees that are possible parallelizable regions when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, it is replaced with a call to the JIT engine. If successfully compiled, a region is transformed to a dataflow graph—a convenient and well-studied computation model amenable to transformation-based optimizations [16]. The instrumented AST resulting from the compilation is finally translated (unparsed) back to shell code and sent over to the underlying shell for execution.

3.3.3 JIT Engine

The PaSh-JIT preprocessor identifies possible parallelizable regions and instruments the shell script to dynamically determine whether they can be optimized by invoking the JIT engine. The JIT engine faces two key challenges: it must not change the original script behavior, and it must run with low overhead as it is invoked multiple times per script.

The JIT engine is a reflective shell script: by inspecting the state of the shell and that of the broader system, it can transparently work with the compiler to determine whether or not to parallelize a script (Fig. 3-2). When running scripts with PaSh-JIT, it is helpful to think of the shell as having two modes: (1) conventional shell mode, where scripts execute in the original shell context, and (2) PaSh-JIT mode, where the runtime reflects on shell state and invokes a compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to PaSh-JIT mode, the JIT engine must carefully save the state of the user's shell; to switch back, it must carefully put things back just the way they were. A shell's state is quite complex: beyond saving and restoring variables, the runtime must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory).

JIT Stages

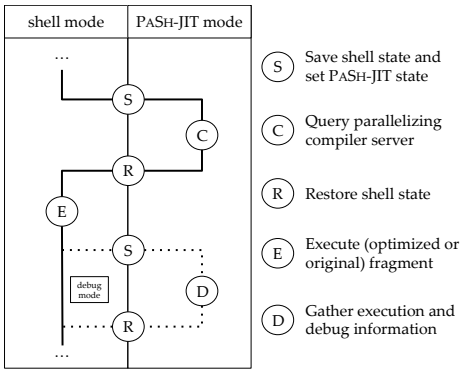

Figure 3-2: Overview of JIT engine stages.

When running normally, the JIT engine transitions into and out of PaSh-JIT

mode once per possible parallelizable region (Fig. 3-2): the JIT engine saves the shell state and switches into PaSh-JIT mode ; then it tries to compile the current fragment (C); whether successful or not, the JIT engine restores the state and switches back to shell mode ; and, finally, either the original fragment or the optimized parallel version is executed (E). With debugging enabled, the JIT engine switches back into PaSh-JIT mode to collect debugging information (D), restoring again afterwards .

Saving : When entering a possible parallelizable region, the first step is to save the shell state—recording the previous command's exit status, the values of environment variables, and the configuration of the shell—essentially, a continuation that can later be restored to execute the target fragment. Once the state is saved, PaSh-JIT mode reconfigures the user's shell to avoid changing script behavior. For example, if the user's shell has the `-e` "exit on error" flag set, the shell should exit immediately when a command (or a pipeline) returns a non-zero exit status, unless that command is in a checked position (*e.g.*, after `!`, or in the condition of an `if` or `while`) [2]. However, failing commands should not stop the JIT itself, so `-e` is unset (and will be restored later in).

Compilation (C): With the state saved and shell reconfigured, PaSh-JIT tries to compile the script fragment: the JIT engine queries the compilation server (§3.3.4) with the script fragment (already parsed during preprocessing) along with the saved shell state, so that the compilation server can try to expand all of the words in the fragment. The server responds to indicate whether it managed to optimize the fragment.

Restoring : Whether or not compilation was successful, the JIT engine exits PaSh-JIT mode, restoring the continuation saved earlier (S) to prepare to execute the fragment. One particular challenge in this mode is to restore state while accommodating different shell modes. Suppose PaSh-JIT is in `-e` mode, trying to run some possible parallelizable region, and the command *before* this region exited with status 47 in a checked position, *i.e.*, without forcing the shell to exit. The JIT engine saves the exit status so as to not overwrite it. The fragment may depend on the exit status, so

PaSh-JIT needs to restore it before running the fragment. But it must be careful—simply running `(exit 47)` would force the shell to exit. Thus PaSh-JIT runs the subshell in a checked position:

```
if (exit "$pash_previous_exit_code"); then
    source "${fragment}"; ...
else
    source "${fragment}"; ...
fi
```

This odd code ensures that the fragment (in identical branches) has access to the previous exit status (in the checked, conditional position of the `if`) without exiting when `-e` is set.

Execution (E): Back in shell mode, the JIT engine executes the fragment. If the compiler was successful, then the JIT engine selects the optimized script fragment. If the compiler failed, the JIT engine falls back to the original fragment. Either way, control flows back to the original shell.

Debug mode (D) : When PaSh-JIT is in debugging mode, the JIT engine will re-enter PaSh-JIT mode after execution (E) in order to log information about the script, such as execution time and exit status. Standard execution skips this extra save/restore cycle.

3.3.4 Compilation server

For each possible parallelizable region, the JIT engine queries the compiler: can this region actually be optimized? To answer this question, PaSh-JIT builds on ideas from the PaSh-AOT [39] dataflow compiler (§3.3.5). As ever, it focuses on preserving behavior and minimizing overhead.

To preserve correct behavior in the face of the shell's dynamism, PaSh-JIT expands each script region prior to compilation.

To minimize overhead due to fixed startup costs—*e.g.*, initialization, dependency loading, logging setup, and output file arrangement—PaSh-JIT packages the new

compiler as a stateful compilation server communicating via Unix domain sockets.[1]

The compilation server is also augmented to support a larger set of optimization opportunities, by storing and using information from one compilation to help another.

The design decision to use a long-lived stateful compilation server has significant implications on performance and the capabilities of the JIT engine:

1. It improves performance and the startup overhead of re-initializing the compiler for each compilation request.

2. It allows us to maintain a state about previous dataflow regions, their status, inputs, outputs, and their execution time. This enabled us to implement features like dependency untangling, and parallelization factor adjustment during run-time.

3.3.5 Command Annotations

PaSh-JIT uses the command annotation and specification framework introduced by PaSh-AOT [39, 16], extended to also indicate whether a command invocation is commutative (§3.4.1). This framework provides information about a command invocation's parallelizability class, inputs, and outputs. A command annotation can be used to extract high-level information about a specific command invocation, *i.e.*, a precise instantiation of its flags, options, and arguments. For example, annotations determine whether a given command invocation is pure and what its inputs and outputs are.

PaSh-JIT uses this annotation framework to extract information for commands that are not shell builtins—that is, commands like `sort` and `grep`. Annotations enable analyses and transformations over command invocations by lifting them to pure dataflow nodes in a dataflow intermediate representation (IR) [16]. For example, `grep -f dict.txt src.txt > out.txt` is a dataflow node with two input files (`dict.txt` and `src.txt`) and one output file (`out.txt`), which are all extracted from

[1] We experimented with both socket and FIFO-based communication, but we saw no significant performance differences.

the annotation of the `grep` command. Annotations also describe parallelization opportunities, *e.g.*, `grep "pattern" src.txt` processes each line of `src.txt` independently, and so it can be parallelized.

3.4 Commutativity Awareness

One of the performance concerns in PaSh-AOT is that when splitting an input across parallel data sections (Fig. 4-2), we need to determine the number of lines to be able to accurately split the stream to multiple output sinks. This is done by making two passes on the input stream. In the first pass, we read the whole input into a temporary file to count the number of lines in the input streams. In a second pass, we split the input stream we stored in temp file equally (by number of line) across the provided sinks. There are two main problems with this approach:

1. In the first pass, we have to write the full input stream from memory into disk; depending on disk speed, this could be a significant hit to performance.

2. When the input is split in the second pass, the data is written sequentially across sinks. This means that for N given sinks, sink i will not start receiving data until all sinks $< i$ completely consume their input. This reduces the amount of parallelism and thus reduces performance.

To address these concerns, we introduced a new splitting model that uses a round-robin strategy for splitting the input stream across the given sinks. Rather than reading the entire input before deciding how to split it, input can be split via small incremental steps that are immediately handed off to data-parallel commands for processing. This splitting model relies on the the commutative nature [32, 22] of many shell commands. Commutative commands can improve parallelization gains by allowing PaSh-JIT to split and process data in small and order-independent batches.

Splitting input into many small batches eliminates both of the problems mentioned above for the following reasons:

1. Constant batch size allows us to split the input stream without knowing the full size ahead of time. This completely eliminates the need to store any temp files on disk.

2. Small batch size increases the pipeline parallelism and improves CPU utilization by overlapping input splitting and processing. Commands across all sinks can start processing their input rather than waiting for the whole input to be read and distributed.

The PaSh-JIT compiler uses these insights to produce more efficient parallel implementations of scripts that contain commutative commands. It introduces a few auxiliary nodes in its intermediate representation (IR) that orchestrate parallel execution for stateless and commutative commands, and compiler transformations that insert these nodes in a dataflow graph. It also provides efficient implementations of these nodes that are instantiated in the parallel target script.

3.4.1 Compilation: Dataflow Model

The PaSh-JIT compiler operates intra-regional compiler builds on the PaSh-AOT compiler, where commands correspond to nodes and communication channels correspond to edges between nodes. To enable commutativity-aware transformations, PaSh-JIT extends PaSh-AOT's annotation framework (§3.3.5) to indicate whether a command invocation is commutative (in addition to its parallelizability characteristics).

Runtime

PaSh-JIT introduces the following four dataflow nodes, which are available as part of PaSh-JIT runtime:

- `c_split`: node takes a single input stream and N output streams. It splits its input into small batches, prepends a header on each batch identifying its sequence number, and then forwards it to one of the N outputs depending

on a load-balancing strategy. Currently, PaSh-JIT implements a round-robin strategy.

- `c_merge`: node performs the inverse operation: it merges N input streams into one and removes any headers.

- `c_wrap`: command is used to wrap stateless commands. It removes the header, forwards the input to the command, and then adds the header back to the command output. forks a new process to execute the wrapped command for each new `block_id`.

- `c_strip`: is a single-input-single-output header-removal node that often precedes commutative commands.

Compiler

We modified the PaSh-JIT runtime to take advantage of the new runtime. We first added a new property to our annotation which indicates if a command is commutative or not. Additionally, all stateless commands are commutative by default. Given the new annotations we modified the dataflow transformations to use the new commutative run-time nodes whenever possible.

To expose the parallelism of commutativity, PaSh-JIT performs the following transformations. The first transformation introduces a pair of `c_split` and `c_merge` before any commutative (*e.g.*, `sort`) or stateless (*e.g.*, `grep`) command.

Another transformation then tries to eliminate unnecessary splits and merges, delaying `c_merge` as far as possible (*i.e.*, enclosing the biggest possible part of the graph). If a stateless command follows a `c_merge`, the command is wrapped with `c_wrap` and the `c_merge` is commuted after it. If a commutative command follows a `c_merge`, the command is parallelized and `c_merge` is transformed to a set of `c_strip`s. Finally, if a `c_split` follows a `c_merge`, then the two are fused together to the identity function, connecting the inputs of `c_merge` with the outputs of `c_split`.

An important execution invariant is that `c_split`s and `c_merge`s (or `c_strip`s) satisfy the requirements of well-formed parentheses, *i.e.*, a `c_split` must always be followed by a `c_merge` or a set of `c_strip`s. PaSh-JIT's dataflow graphs are essentially bimodal, since subgraphs that are between `c_split`s and `c_merge`s will execute with batches, requiring all commands in them to be wrapped with `c_wrap`, while the rest of the dataflow graph executes like the original.

3.4.2 Runtime: Commutativity Implementation

For commutativity to show performance improvement over the previous approach, the runtime needs to be highly efficient. The cost of (1) forking new processes and (2) adding and removing headers for every mini-batch need to be almost non-existent. Otherwise, constant overheads dominate over any performance gains we get from commutativity. For this reason, we implemented the runtime in C and paid careful consideration to all aspects of the implementation. This involved things like minimizing buffer copying, choosing the optimal internal buffer size, and using non-blocking pipes when possible. The following sections describe some the optimizations and special considerations we had to implement to maintain high performance across many input sizes and situations.

Protocol

To reconstruct the order of different outputs while merging, PaSh-JIT needs to keep track of the ordering as input batches are sent to different command copies for processing and, more generally, as input-output batches flow throughout the parallelized script. To achieve this, PaSh-JIT wraps all input batches with a header that contains the three following fields: `block_id`, for ordering blocks; `block_size`, the size of the block in bytes; and `is_last`, a boolean value, set to true only for the last block with a given `block_id`.

```
struct header {
    int block_id;
```

```
    uint64_t block_size;
    bool is_last;
}
```

Utilization and deadlocks

PaSh-JIT must avoid deadlocks during write operations between the wrapper commands and the commands they wrap—*i.e.*, the two should never be blocked trying to write at the same time. Additionally, the wrappers must maximize utilization of the command they wrap, *i.e.*, they should never wait on input unnecessarily. To avoid deadlocks, PaSh-JIT wrappers use non-blocking read and write; and to increase utilization and reduce waiting time, they write in small chunks of 32KB.

Handling inputs with long lines

An input may contain lines that are longer than the `c_split` block size. Such an event leads to non-uniform block sizes and high memory consumption, because each block must be read and sized completely before splitting and adding to the header. PaSh-JIT addresses this issue by introducing the `is_last` header field in every batch: if a block exceeds the specified size due to containing large lines, the block is split into multiple blocks; all blocks share the same `block_id` but only the last sets `is_last` to true. All blocks with the same id are forwarded sequentially which means we don't need to keep track of of the order within the same id. Downstream commands can then use the `is_last` information to correctly reconstruct the output. Block splitting reduces memory requirements and improves performance, as it allows for higher utilization regardless of the frequency of newlines. And blocks maintain a constant size throughout the flow, despite the presence of commands with high output-to-input ratios, such as `curl`.

Handling small inputs

Inputs that are smaller than `c_split`'s block size lead to a single block and thus sequential execution. PaSh-JIT's `c_split` addresses this issue by first attempting

to read an input size s equal to `sink_count * block_size` bytes before forwarding any blocks. If the total input is larger than s, this buffering ensures that all parallel instances will get at least one block; if the total input is smaller than s, then the input read is re-split into blocks fairly and forwarded downstream. The size s is configurable and defaults to 1MB, which we empirically determined avoids both high overhead and low utilization.

An example of the output of the compiler with commutativity enabled can be seen below.

```
c_split /tmp/fifo8 /tmp/fifo9 /tmp/fifo10 &
c_wrap 'grep "^....$"' </tmp/fifo9 >/tmp/fifo11 &
c_wrap 'grep "^....$"' </tmp/fifo10 >/tmp/fifo12 &
c_strip </tmp/fifo11 >/tmp/fifo13 &
c_strip </tmp/fifo12 >/tmp/fifo14 &
sort </tmp/fifo13 >/tmp/fifo15 &
sort </tmp/fifo14 >/tmp/fifo16 &
eager.sh </tmp/fifo15 >/tmp/fifo17 &
eager.sh </tmp/fifo16 >/tmp/fifo18 &
sort -m /tmp/fifo17 /tmp/fifo18 >/tmp/fifo19 &
```

3.5 Dependency Untangling

The design of the JIT compiler makes it possible to maintain a state across data regions. The compiler uses this additional state to make decisions of whether it is safe to run the following data region in parallel with other data regions that are still running. A direct example of this is for-loop bodies, when the compiler is called on iteration i, it knows the status (compiled successfully, done, still running) and the input and outputs of all previous iterations and thus can make decision on whether it is safe or not to run the next iteration in parallel. To do this, the compilers checks for (1) all currently regions have been compiled successfully and thus have no side effects and (2) the inputs and outputs of all currently running iterations have no conflict

with the dataflow under consideration.

3.6 Evaluation

In this section, I focus on the performance evaluation for Commutativity (§3.6.1) and Dependency Untangling (§3.6.2). For this evaluation, we used a variety of benchmarks and workloads that expose both cross-regional and intra-regional parallelization opportunities (Table 3.1). The complete evaluation of PaSh-JIT correctness and performance in comparison to PaSh-AOT and Bash can be found in the PaSh-JIT paper [20].

Table 3.1: **Benchmark summary.** Summary of benchmarks used to evaluate Commutativity and Dependency Untangling performance for the thesis

	Benchmark Set	Short Label	Sections	Scripts	LOC	Input	Source
1	Common & Classic One-liners	Classics	§3.6.1	10	123	14G	[6, 5, 36, 19, 25]
2	Bell Labs Unix50	Unix50	§3.6.1	36	142	21G	[23, 7]
3	Microbenchmarks	MicroBench	§3.6.1	1	6	—	custom
4	COVID-19 Transit Analytics	COVID-mts	§3.6.2	4	79	3.4G	[38]
5	Natural-Language Processing	NLP	§3.6.2	21	306	1060 books	[10]

Hardware & software setup: The evaluation was run on a ThinkStation P340 machine with 20 physical × 2.80GHz Intel(R) Core(TM) i9-10900 CPU with 32GB of Ram, Ubuntu 18.04.

3.6.1 Commutativity Awareness

To evaluate the benefits of commutativity, we focus on scripts with individual region parallelization potential: Classics, Unix50, COVID-mts, and AvgTemp. Toggling on the commutativity optimization achieves an average speedup of 1.43x and a maximum speedup of 2.43x compared to toggling it off. Fig 3-3 shows the speedup in relation to Bash as a baseline.

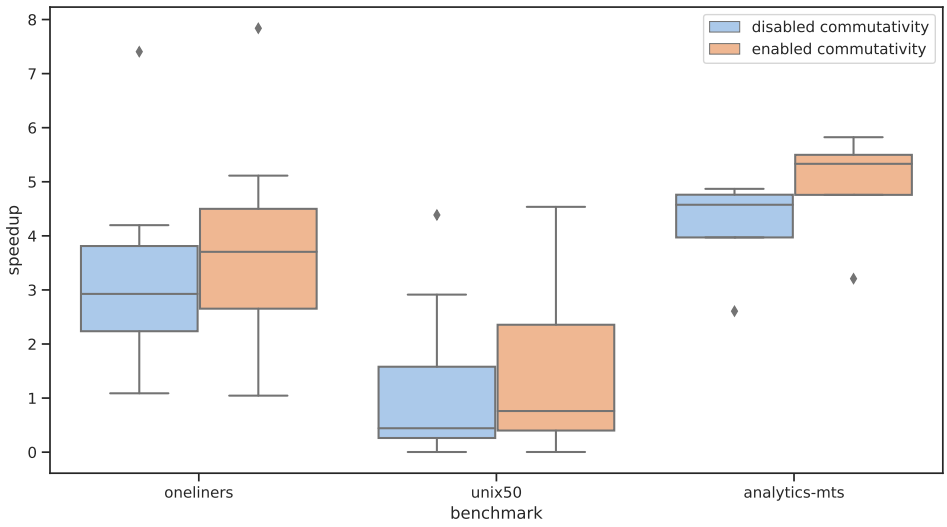

Figure 3-3: Commutativity speedup over Bash baseline

3.6.2 Dependency Untangling

To evaluating Dependency untangling we focused on scripts with cross-region parallelization potential. Independent for-loop iterations are ideal candidates for cross-region parallelization. Turning on the dependency untangling optimization achieves an average speedup of 3.43x and max of 4.42x on NLP in comparison to toggling it off (Fig. 3-5). The average speed-up of PaSh-JIT is 1.94x with dependency untangling enabled and 0.64x when disabled (Fig. 3-4). The reduction in performance with dependency untangling is caused by constant overheads from invoking the JIT compiler over tight for-loop iterations.

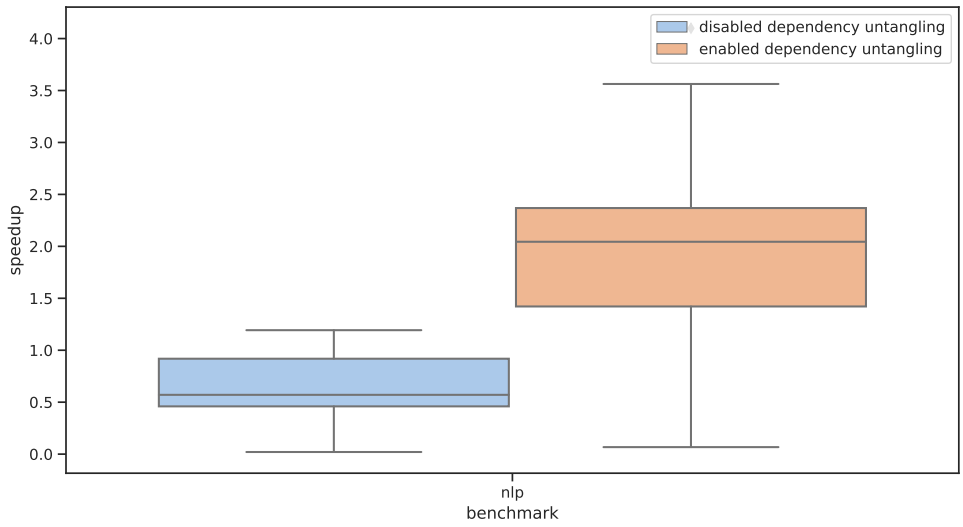

Figure 3-4: Dependency Untangling Speedup over Bash baseline

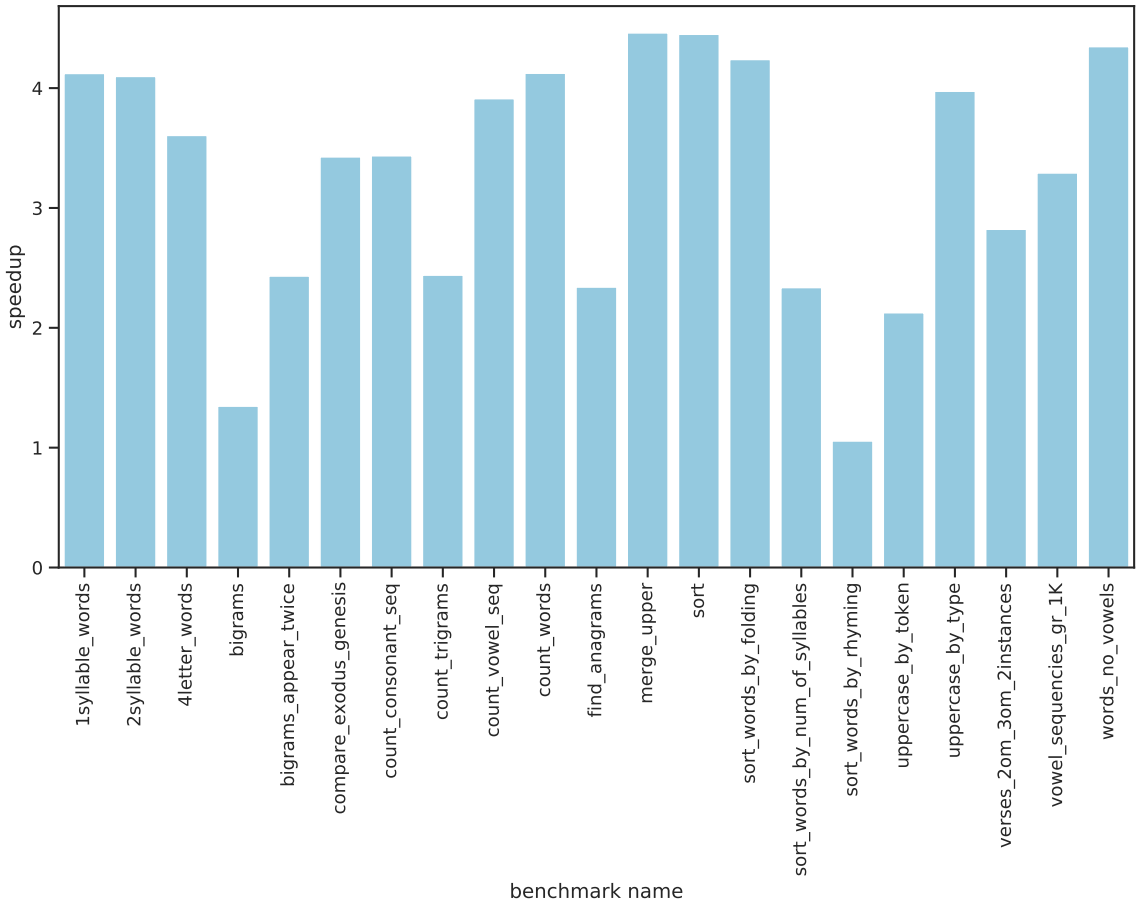

Figure 3-5: Dependency Untangling Speedup on NLP benchmarks

Chapter 4

Distributed-PaSh

4.1 Intro

The move from parallel to distributed execution seems like a natural progression; however, the use cases and trade-offs of PaSh-JIT change fundamentally when moving from the one-machine model to the distributed model. This system faces three key challenges: it should maintain the same behaviour as local execution, it should require no or very minimal changes to the script, and finally the distribution benefits should overweight the high network overheads compared to parallel execution.

In this chapter, I present Distributed-PaSh, the first distributing JIT compiler for the shell. Distributed-PaSh aims to efficiently utilize cluster's resources across multiple dimensions such as computing and network resources, as well as data locality. Distributed-PaSh blends the parallel and distributed execution models to achieve higher performance than what's possible by each model individually. In §4.2, I explain the design for executing shell scripts in a distributed fashion. In §4.3 I explore the integration of distributed file systems (HDFS in particular) into our system. This allows Distributed-PaSh to make decisions based on data locality with respect to how to distribute and run parallel data pipelines. In heavy data processing scripts, Distributed-PaSh can achieve higher performance than single machine sequential and parallel shells like Bash and PaSh-JIT.

4.2 Distributed Execution

4.2.1 Design

We wanted the design to naturally fit on top of PaSh-JIT. The new system should be able to leverage the power of PaSh-JIT since the benefits of the JIT engine are relevant to distributed execution. The JIT engine's ability to make parallelization decisions closer to the point of execution is critical for correctness. Other optimizations, such as commutativity awareness and dependency untangling, are even more relevant in the distributed model. Distributed-PaSh could unroll a for-loop and run iterations in distributed fashion, which would allow for higher utilization of cluster compute resources and could lead to significant speed-ups if the for-loop body is doing a heavy computation. Furthermore, commutativity allows Distributed-PaSh to efficiently utilize network resources by sending small chunks of data across machines in round-robin fashion.

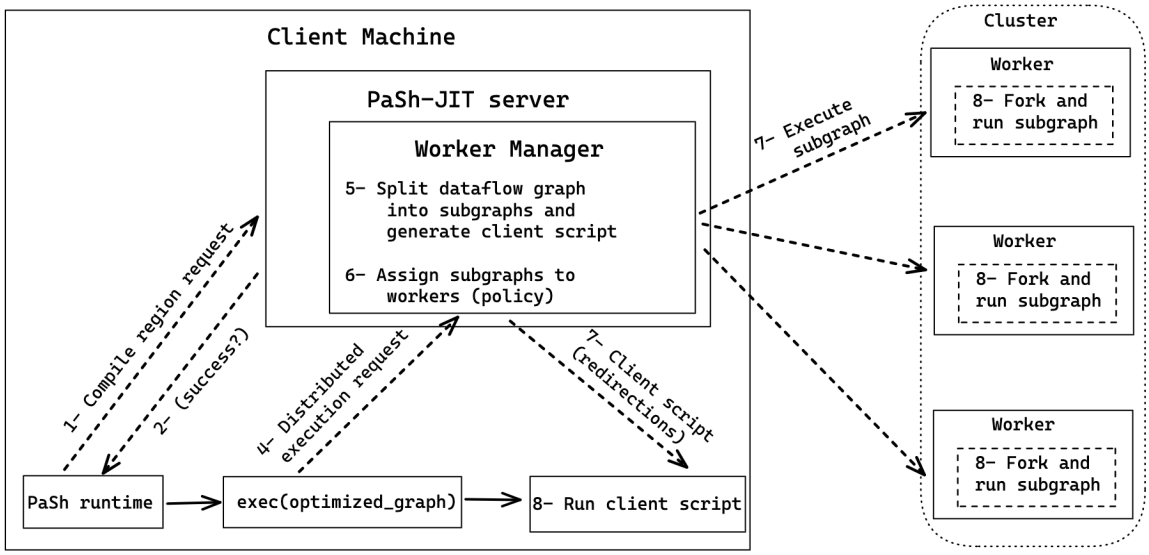

Figure 4-1: Distributed PaSh design

4.2.2 Components

Distributed-PaSh has the following components:

- **PaSh-JIT:** The JIT compiler as described in 3.3 the pre-processing and com-

pilation of the dataflow regions.

- **Worker Manager:** Takes the optimized dataflow graph and divides it into different subgraphs that are sent to available workers in the cluster.

- **Workers:** The primary unit of execution. A worker accepts execution requests from the worker manager. An execution request contains a dataflow graph objects which the worker compiles into a shell script and runs on its local machine. The design decision to transfer dataflow graph instead of ready-to-run shell script allows us to dynamically generate any required temp files without needing any prior information about the worker's particular file system layout. This also allows us to run a second round of parallelization on the subgraphs themselves, making it possible to take advantage of the workers' additional cores.

4.2.3 System Setup and Assumptions

We assume that all machines in the cluster are able to communicate freely on any port. The system administrator needs to do an initial setup to install the DISTRIBUTED-PASH run-time and establish a worker process once on every machine in the cluster. We also assume that the users, servers, and network are trusted; Further work can improve security through isolation, encryption, and other methods.

We also do not implement any fault tolerance mechanism in the system; adding fault tolerance to this system would require a fair amount of complexity that we could not justify. The user can simply rerun the script in the rare occasion a worker dies or network partition happens. We believe that this level of fault tolerance is acceptable for most users who will be using this system.

4.2.4 Runtime and Implementation

Workers

Each machine intended to be used should have a long-lived worker running. The worker receives requests to execute dataflow graphs from the worker manager. Every request contains a serialized dataflow graph object and any functions defined or exported in the shell script. The worker compiles the dataflow graph into a shell script and then runs it after importing the supplied functions. The worker also runs a **Discovery service** in a different process. The discovery service helps the read and write ends of the distributed pipes identify each other. The discovery service is implemented as a gRPC server with two RPC calls, a put RPC which stores an mapping between an key and address in an in memory hashmap and a get RPC which retrieves the address for a given key.

Worker Manager

The worker manager sits next to the PaSh-JIT compilation server. When the compilation server receives a request from the JIT engine to compile a dataflow region, the compiler optimizes the region, and if compilation succeeds, it gets sent to the worker manager. The worker manager can safely split the parallel dataflow graph into multiple subgraphs and then distributes these subgraphs to workers for execution. The manager learns about the workers in the cluster using a config.json file defined in the Distributed-PaSh home directory. This cluster configuration file can be extended to include things like the number of cores on each machine. This could be utilized by the manager to more efficiently distribute work. The worker manager performs two key tasks:

1. Graph splitting: The graph splitting algorithm tries to split every continuous parallel and sequential section in the graph into its own subgraph figure 4-3. We replace some of the previous UNIX pipelines with our implementation of distribute UNIX pipes so that subgraphs can move data across machines. A couple consideration need to be made for correct execution and to maintain local shell

behavior. In particular, named fifo, standard input and output, and files are usually relative to the directory where the shell was executed from and are local to the machine. We should be able to handle them without affecting correctness and without making assumptions about the file system or working directory of the client or workers. To solve this problem, we generate an additional script that runs directly in the shell where the user ran DISTRIBUTED-PASH; This script handles redirection to and from local files, FIFOs, and standard input and output.

2. Work distribution: The worker manager distributes the work to workers according to a defined policy.

Distributed Pipe

We introduced a new runtime primitive to simulate a UNIX data pipes over a network. The distributed pipe was implement using go and it is essentially a wrapper around sockets. It includes a gRPC client that connects to the discovery service to query and publish the read and write ends of the socket. In detail, a distributed pipe takes the following arguments.

- Type: Determines whether the node is at the read or write end of the pipe. This is analogous to the read and write ends of a normal UNIX pipe.

- UID: Every distributed pipe has a unique identifier. The read and write ends of the pipe use the same UID.

- Discovery Service Address: The read and write ends of a distributed pipe identify each other using the UID. In particular, a write publishes an address represented by a host and a port number to the discovery service. The read end then queries the discovery service to get the address to read the data from. If the discovery service doesn't have a particular UID, it returns an error. This is common, as the write end could take a little longer to publish its address. The read end retries for 10 seconds before failing.

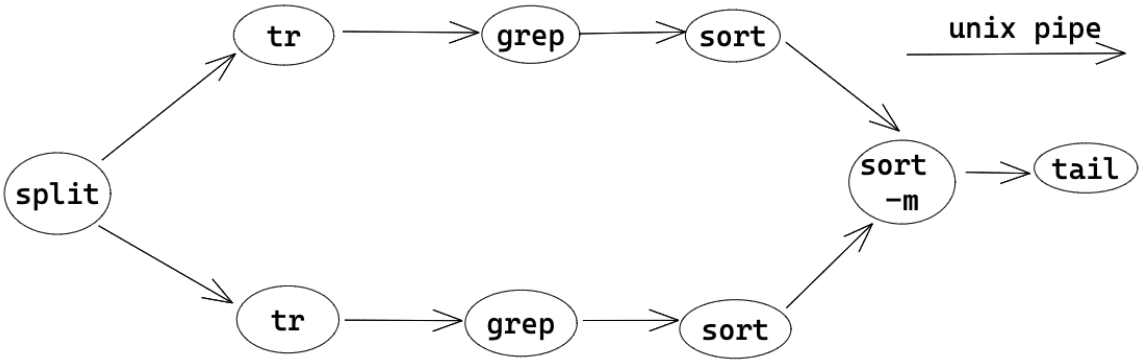

Figure 4-2: Dataflow graph before splitting

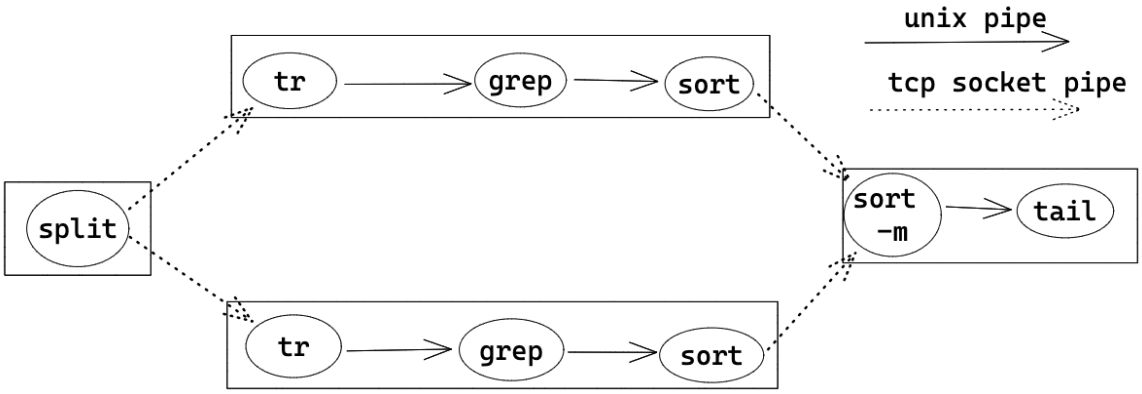

Figure 4-3: Dataflow graph after splitting

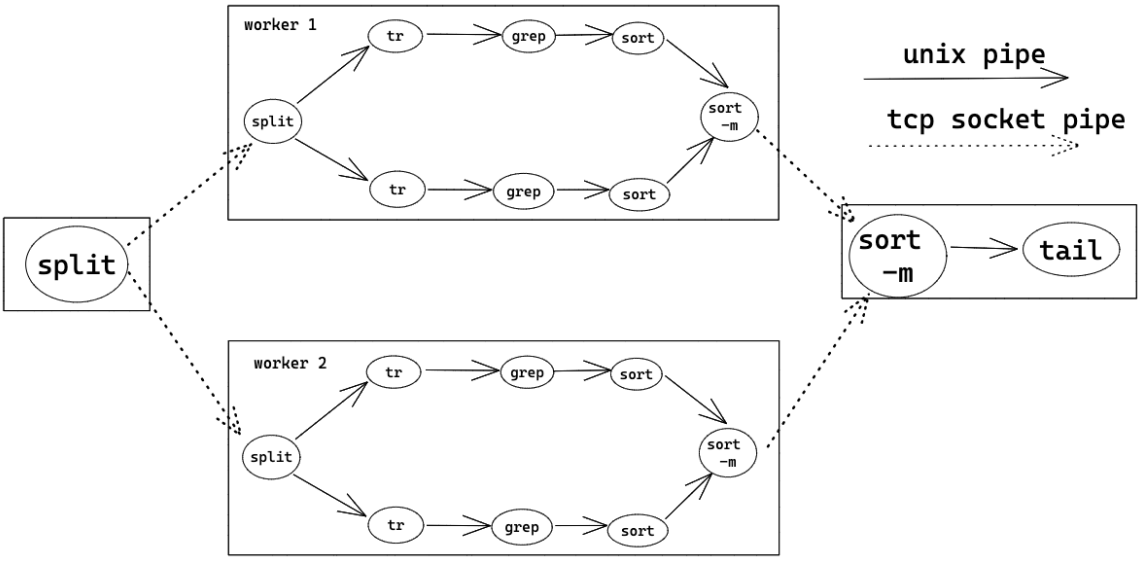

Figure 4-4: Second round of parallelization on worker's machine

4.3 Distributed File Systems

Interacting with the local file system is the heart of the UNIX shell and one of its primary functions. Most shell commands interact directly with the file system to read, store, and process files. When we started working Distributed-PaSh, we had to give special attention to how user interactions with the file system change when moving from the one machine model to the distributed model. We quickly realized that distributed file systems should be first-class citizens in a distributed shell. We decided to pursue a design that is general enough to incorporate most distributed file systems. However, in our implementation, we decided to focus on the Hadoop Distributed File System (HDFS). In the following sections, I give a brief overview of NFS and HDFS and why we choose to primarily focus on HDFS in our implementation. Then I describe our design and the implementation for a data-locality-aware Distributed-PaSh.

4.3.1 Background

The two most common distributed file systems we came across were the Network File System (NFS) and the Hadoop Distributed File System (HDFS). Both of these file systems are widely used in distributed clusters but their capabilities and use cases are fundamentally different.

NFS

The Network File System is a protocol that was developed by Sun Microsystems in 1984 to allow a client to view store and update files over a computer network. NFS primary purpose is to allow users to access and modify files on other machines as if the file was on the local machine. NFS does not provide any replication for files, as it is primarily an interface to access another computer disk.

HDFS

The Hadoop Distributed File System is designed to be run on commodity networked computers or nodes. The file system provides fault tolerance by replicating the file across multiple nodes. This is done by splitting the file into blocks of configurable size (usually 125 Megabytes) and then replicating the blocks across multiple machines. The default replication factor is 3 and block size is 125 Megabytes; both of these can be configured for the whole file system system and/or for individual files.

In this work, we focus on HDFS instead of NFS for the following reasons:

- The HDFS filesystem stores files by splitting them into blocks and distributing these blocks across many nodes. This property fits very well with PaSh dataflow parallelization model which usually starts by splitting the input stream (figure 4-2).
- HDFS file replication provides fault tolerance, but that is only one benefit. Replication reduces the network, disk, and processor usage on any particular machine and therefore significantly improves performance as file block can be fetched from any machine that contains a replica of that block. This solves many of the common bottlenecks NFS suffers from, especially when multiple users access the same file.
- HDFS is commonly used in big data processing alongside frameworks such as the Hadoop Map-Reduce framework.

4.3.2 Design

The network is usually one of the first bottlenecks that appears when dealing with large files, and it is usually difficult for developers to address it without significant changes to their infrastructure and code. Additionally, a common pattern in many shell scripts and map-reduce jobs is that data size gets smaller as we progress in the dataflow graph. This usually means that delaying data movement until a later stage

would usually lead to moving less data across machines. We decided to augment our design in 4.2 to take these factors into consideration when distributing work across workers.

Let's take the following the following script:

```
hdfs dfs -cat $IN | tr A-Z a-z | grep foo | sort
```

This script reads the file `IN` from HDFS, converts all letters to lower case, then greps the word foo, and finally sorts the output. Executing this script either in Bash or PaSh-JIT would require the the HDFS cat to move the whole file over the network for this script to fully execute. The cost of transferring files over a network could be substantial, in our experimental cluster we measured around 280s for just executing the first HDFS cat command while executing the file script took around 290s. This shows that most of the execution time was spend on transferring the file across the network. The size of the data after grep is likely to be orders of magnitude smaller the file itself and the commands `tr` and `grep` are stateless commands which PaSh-JIT can easily parallelize (Fig. 4-2). Distributed-PaSh improves the performance this script by (1) executing the as much as possible on the machine containing the data and (2) directly operation on the pre-split HDFS file blocks and thus eliminating the bottleneck of having to merge all the blocks into one file.

Remote File Resources

To integrate distributed file systems into our distributed execution model, we introduced a new type of file resource. Previous file resources consisted of (1) file resources that represents files on a user's local machine, (2) file descriptor resources representing named pipes, and (3) ephemeral resources representing anonymous pipes. We added a remote file resource, which primarily includes additional information about the location of the data and supports having multiple locations in case of replication. This allows the worker manager to take data location into consideration when assigning jobs to workers.

Design benefits

The augmented design allows Distributed-PaSh to efficiently utilize cluster resources on three primary dimensions:

1. Disk resources: Executing tasks on workers who have the data locally leads to tasks finishing faster due to data locality and no networking overhead.

2. Networking resources: The network traffic overall since shell scripts and map-reduce jobs output is much smaller than the input.

3. Processing resources: Distributed-PaSh can parallelize the graph sections as in figure 4-4 and therefore can take advantage of both the cluster nodes and the cores on every node.

Distributed-PaSh is able to do this without requiring developers or casual users to make any modifications to their scripts. Distributed-PaSh can achieve significant improvements of up to 8.17x over Bash and 4.65x over PaSh-JIT in our benchmarks. We expect these results to greatly vary based on network bandwidth and condition, node computing power, type of disk, and other factors. However, we expect Distributed-PaSh to be faster than running the script in the Bash shell when dealing with large amounts of data that is located on a distributed file system.

HDFS integration

Our design assumes the following:

- The user has access to HDFS by either being part of the cluster nodes or through a client.

- The nodes in the cluster have Distributed-PaSh runtime installed and workers running.

When the JIT compiler is operating on a dataflow region, we check if the input file resource is an HDFS file. When we detect an HDFS file resource, we query HDFS for the location of the file blocks and initialize remote file resources for each block.

We then create a stream per block which effectively replaces what used to be the split node in PaSh-JIT (Fig. 4-2). This does not affect the semantics and correctness of the PaSh-JIT compiler. The remainder of the execution goes through the same process described in 4.2 with a slight change in the worker manager policy.

Policy

The policy for determining which worker would operate on which part of the graph was augmented to take into account the location of the file resources. If a worker is available where the data is, we execute the task on that worker; if the data is replicated and multiple workers have these data blocks on their local disks, we choose the worker with the least amount of work.

4.3.3 Implementation

Remote File resources

The `RemoteFileResource` described above is an interface that includes an addition method `is_available_on(host)`. The method should return True or False depending on whether the resource is available on a given host. Every remote file resource would implement this interface and method.

Every HDFS blocks is represented by `DFSSplitResource` which is a class that implements `RemoteFileResource`. This class contains information about the primary block it is representing and other blocks in the file. This is necessary since HDFS doesn't always split a file on newline characters which would require us to fetch a small part of the following block sometimes.

DFS File Reader

The file reader is a grpc service that is part of the worker on every machine. The primary function of the file reader is to read the local file or block from the worker's disk given a path. The file reader server and client are implemented in Go and communicate through protocol buffers (protobufs). The RPCs of the main server, the

client functions and the configuration structure are described in listing 1. The client takes as input a configuration describing the blocks of the file and their locations, a primary split number if the file consists of more than one block, and finally a prefix if the file path needs additional custom prefix based on the worker machine. The config structure is flexible enough to represent files that are stored in various ways including HDFS and NFS. A file stored on NFS can be described with a configuration consisting of one block and one host, while a file on HDFS would have many blocks and many hosts per block. The file reader also makes sure that every block is terminated with a newline character, this is done by reading logical splits instead of the raw block. A logical split is a block starting at the first full line and ending at a new line character. The file reader achieves this by reading up until the first newline character in the next block. The full algorithm for reading a block is described in Appendix A (listing 2).

```
Config Structure : []Blocks {
            Path string // a string representing the block path.
            Hosts []string // an arrays of addresses containing this block.
            }
```

Listing 1: File Config Structure

4.4 Evaluation

4.4.1 Experiment Setup

The experiment setup consists of a cluster of 4 nodes. For ease of deployment, we used docker swarm to deploy HDFS and the Distributed-PaSh runtime on a 4 machine swarm. All machines are identical, with 6 core Intel(R) Core(TM) i7-10710U cpu and 1 terabyte of SSD. All machines were located in the same room and we measured the bandwidth between them to be 90.8 Mbits/sec. The benchmarking of single machine shells (Bash & PaSh-JIT) was done on a machine with 20 physical × 2.80GHz Intel(R) Core(TM) i9-10900 CPU with 32GB of Ram. All benchmarks were run in Ubuntu 18.04 docker containers.

4.4.2 Benchmarks

To evaluate Distributed-PaSh we minimally modified the scripts in Classics to take input from files hosted on HDFS instead of local disk. The average input size to the scripts was around 3GB. We measured the execution time of each test using Bash, PaSh-JIT, and Distributed-PaSh.

4.4.3 Results

Distributed-PaSh achieves an average of 3.34x and a max speedup of 8.17x over Bash (Fig. 4-5). Additionally, since Distributed-PaSh performs both parallelization and distribution of parallel dataflow pipelines, we benchmark it against PaSh-JIT to isolate the performance improvements of distribution. Distributed-PaSh achieves an average of 1.64x and a max of 4.65x speedup over PaSh-JIT (Figure 4-6). Only one scripts from Classics (diff) had worse performance than using Bash and PaSh-JIT; the script utilizes named FIFOs in its computation, which requires Distributed-PaSh to move large amounts of data back and forth between the cluster machines and the client machine (to preserve local shell semantics). Improved heuristics could potentially improve this scenario.

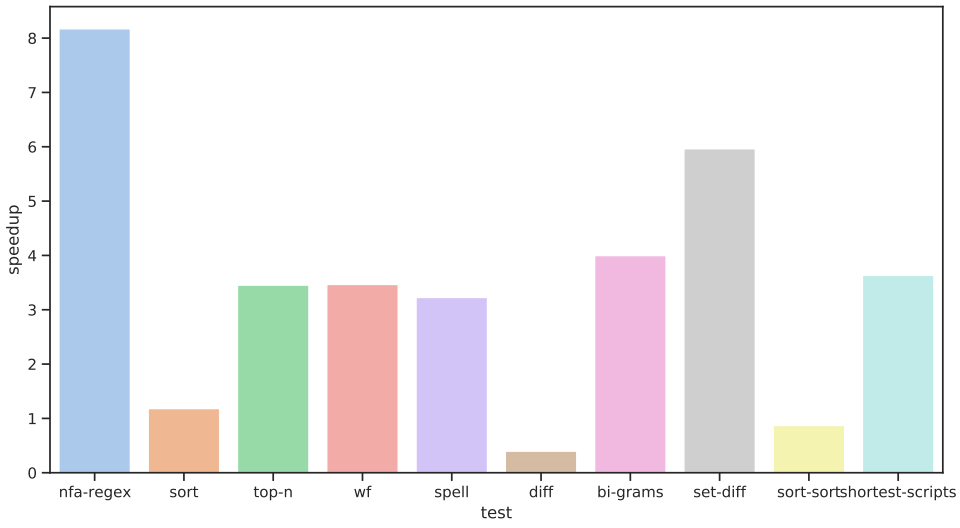

Figure 4-5: Speedup over Bash

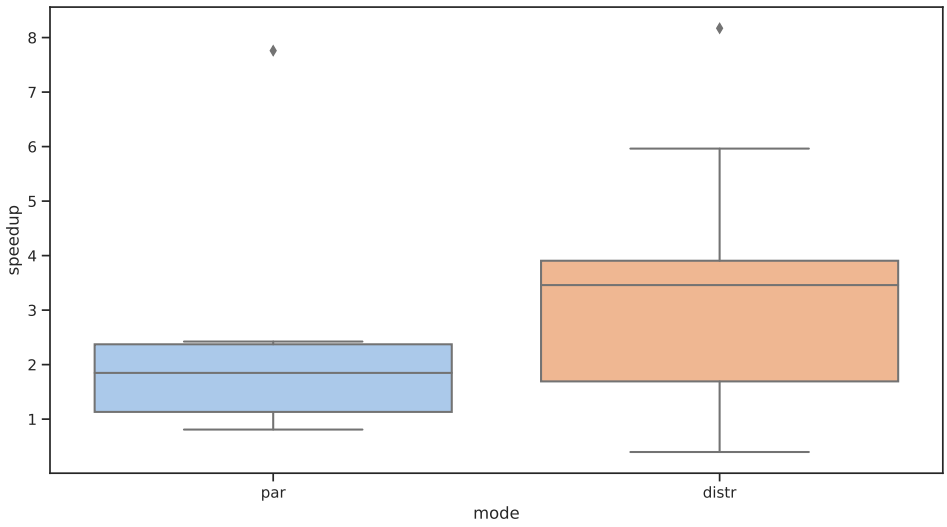

Figure 4-6: Speedup over Bash

Chapter 5

Conclusion

5.1 Summary

In this thesis, I explored the current state-of-the-art shell parallelization and distribution methods. I described the PaSh-JIT system and my contribution to it. I also presented Distributed-PaSh, the first distributing JIT compiler for the shell. I explored the design and implementation of this system and showed that such system is capable of significantly improving the performance of shell scripts in distributed environments.

5.2 Future work

This work serves as a foundation for advancing the capabilities and performance of the UNIX shell. The integration of distributed pipeline execution and a JIT compilers remains relatively unexplored. There is still work to be done in optimizing, defining and implementing the right abstractions, features, and tools for the distributed model. Other extensions, such as fault tolerance and security could be explored. In addition, this work could serve as a foundation adapting the shell to new and emerging computing models. I detail some ideas for future work:

5.2.1 Fault Tolerance

Distributed-PaSh doesn't implement any form of fault tolerance. It is not clear what level of fault tolerance we should aim for; we haven't defined what situation we would like to recover from and what situation we should consider unrecoverable. Recovering when a server machine dies in middle of execution would require some form of check-pointing in addition to careful control and possibly duplication of data flowing to and from, files, named pipes, standard input and output. In addition, recovering a server running stateless commands could be done more easily than recovering servers running stateful commands, the latter might require a maintaining a mirror which we can switch to in case of failure.

5.2.2 Security

The current system assumes that both the client is trusted, realistically this might not always be the case. Some for of containerization, access control, and stream encryption could help elevate most security concerns.

5.2.3 Data driven decision making

Currently we don't collect any health metrics in our implementation. It might be useful to insert nodes that measure the dataflow and can report if a machine is being slower than other machines either to having slower network connection or strained resources. The worker manager can take this data into consideration when distributing future work. This data can also be used for checking the health of the workers which would help us catch bugs such as leakages causing disk or compute over utilization.

5.2.4 Optimistic execution

Distributed execution comes with heavy costs. The overheads communicating and moving data between processes could be very large and it is much more pronounced than in the parallel execution model. In addition, network speed and bandwidth

could be very inconsistent in comparison to single machine communication methods. In some scripts, these overheads dominate any benefit that could be gained from running distribution and sometimes even from parallelization. A possible remedy for optimizing scripts is to optimistically execute the sequential script during the compilation stage of the parallel or distributed scripts.

5.2.5 Policy

The current work distribution policy uses some simple heuristics to determine the best worker. In addition to the data-driven approach mentioned above (§5.2.3), it is possible to improve the policy from a higher-level perspective. The characteristics of distributed clusters, such as network throughput, compute power, and predicted usage patterns vary greatly; developing abstractions and tools to improve work allocation would be beneficial not only for Distributed-PaSh, but also for large-scale data processing systems in general.

5.2.6 Serverless PaSh

Research about adapting the shell to new and emerging computing models such as serverless is still in its early stages. The work done in PaSh-JIT and Distributed-PaSh could aid in the design and development of a serverless shell that allows users to utilize the power of serverless computing seamlessly.

5.2.7 Traditional distributed shell

It could be interesting to create a shell that mixes between the traditional distributed shells used for system administration and Distributed-PaSh. This would involve adding an interactive interface in addition to any other features that those shells typically provide.

Appendix A

Listings

```
func read_logical_split(config, block_num):
    skip_first_line = True
    local_file_reader = FileReader(config.path[block_num])

    if block_num == 0: // First block doesn't skip the first line
        skip_first_line = False

    if skip_first_line:
        err = local_file_reader.seek_to_delim('\n')
        if err == EOF: // This block belongs to previous logical split
            return

    Stdout.write(local_file_stream) // Write remaining

    for block in blocks[block_num + 1]:
        // read the first line of the following block
        err = read_first_line(block, Stdout)
        if err == EOF: // no newline -> keep reading
            continue
        else:
```

```
break
```

Listing 2: Algorithm for reading file logical splits

Bibliography

[1] Apache Software Foundation. Hadoop.

[2] The Austin Group. Posix.1 2017: The open group base specifications issue 7 (ieee std 1003.1-2008), 2018.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

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

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

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

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

[8] Carl Friedrich Bolz. *Meta-tracing just-in-time compilation for RPython*. PhD thesis, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2014.

[9] Craig Chambers. Staged compilation. *ACM SIGPLAN Notices*, 37(3):1–8, 2002.

[10] Kenneth Ward Church. Unix™for poets. *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*, 1994.

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

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

[13] Google. V8 javascript engine. `https://developers.google.com/v8/`.

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

[15] Michael Greenberg. libdash. `https://github.com/mgree/libdash`, 2019. [Online; accessed December 6, 2021].

[16] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[17] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, 2000.

[18] Neil D Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

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

[20] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, just-in-time shell script parallelization. In *16th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 22), pages 1–18. USENIX Association, July 2022.

[21] Konstantinos Kallas and Konstantinos Sagonas. Hiperjit: A profile-driven just-in-time compiler for erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pages 25–36, 2018.

[22] Deokhwan Kim and Martin C Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 528–541, 2011.

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

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

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

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

[27] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

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

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

[30] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, 1998.

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

[32] Martin C Rinard and Pedro C Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

[33] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. corr abs/2012.15443 (2021). *arXiv preprint arXiv:2012.15443*, 2021.

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

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

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

[37] Scott Thibault, Charles Consel, Julia L Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[38] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens. `https://bit.ly/3s112R5`, 2021.

[39] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing.

In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

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