POSTER: Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat

Jiasi Shen
MIT
USA
jiasi@csail.mit.edu

Martin Rinard
MIT
USA
rinard@csail.mit.edu

Nikos Vasilakis
MIT
USA
nikos@vasilak.is

Abstract

We present KumQuat, a system for automatically generating data-parallel implementations of Unix shell commands and pipelines. The generated parallel versions split input streams, execute multiple instantiations of the original pipeline commands to process the splits in parallel, then combine the resulting parallel outputs to produce the final output stream. KumQuat automatically synthesizes the combine operators, with a domain-specific combiner language acting as a strong regularizer that promotes efficient inference of correct combiners. We present experimental results that show that these combiners enable the effective parallelization of our benchmark scripts.

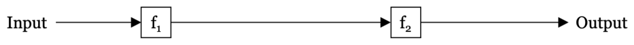
CCS Concepts: • **Computing methodologies** → *Concurrent computing methodologies*; • **Software and its engineering** → *Software development techniques.*

Keywords: Automatic parallelization, program synthesis

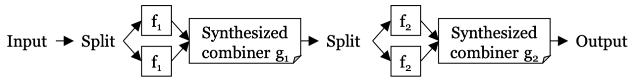
1 Introduction

The Unix shell, working in tandem with the wide range of commands it supports, provides a convenient programming environment for many stream-processing computations. Shell commands—which can be written in multiple languages—typically execute sequentially on a single processor. This sequential execution often leaves available data parallelism, in which a command operates on different parts of an input stream in parallel, unexploited. This observation has motivated the development of systems that exploit data parallelism in shell pipelines [5, 10, 14]. A key prerequisite is obtaining the combiners necessary for merging the resulting multiple parallel output streams correctly into a single output. Previous systems rely on developers to manually implement such combiners and associate them with their corresponding shell commands [5, 10, 14].

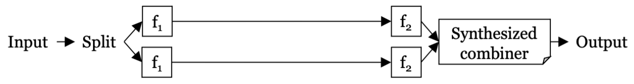
PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508400



(a) Serial pipeline that consists of two commands f_1, f_2



(b) Unoptimized parallel pipeline that executes a combiner after each parallel command



(c) Optimized parallel pipeline that executes multiple commands in parallel after eliminating intermediate combiners

Figure 1. KumQuat parallelizes a pipeline by splitting input streams, running parallel copies of original commands on the input substreams, and combining output substreams.

We present a new system, KumQuat, for automatically exploiting data parallelism available in Unix pipelines. Working with the commands in the pipeline as black boxes, KumQuat automatically generates inputs that explore the behavior of a command to infer and automatically generate a combiner for it. This capability enables KumQuat to automatically generate data-parallel versions of Unix pipelines, including pipelines that contain new commands or command options for which combiners were previously unavailable.

KumQuat targets commands that can be expressed as data-parallel divide-and-conquer computations with two phases:[1] the first phase executes the original, unmodified command in parallel on disjoint parts of the input; the second phase combines the partial results from the first phase to obtain the final output. A domain-specific combiner language acts as a strong regularizer that promotes efficient learning of correct combiners. The resulting (automatically generated) parallel computation executes directly in the same environment and with the same program and data locations as the original sequential command.

[1]There is no requirement that the actual internal implementation must be structured as a divide-and-conquer computation—because KumQuat interacts with the command as a black box, the requirement is instead only that the computation that it implements can be expressed in this way.

2 Working Example

Consider a Unix shell pipeline that counts the frequency of words in an input stream [2]:

```
cat $IN | tr -cs A-Za-z '\n' | tr A-Z a-z |
    sort | uniq -c | sort -rn
```

The six commands in the pipeline (1) read the input stream, (2) break the input stream into lines of words, (3) convert words into lower case, (4) sort words, placing each word on its own line, (5) compare adjacent words, removing duplicates and prepending each unique word with a count, and (6) sort the words on their counts in reverse order.

KumQuat exploits the data parallelism available in each command by splitting the input stream into substreams and instantiating copies of each command to process the substreams in parallel. These parallel computations produce a set of parallel output substreams. A key question is how to appropriately combine these output substreams into a single final output stream of the command. KumQuat synthesizes a combiner for each parallelizable command (Figures 1a,1b).

To focus synthesis on a productive space of candidate combiners, KumQuat works with combiners expressible in a domain-specific combiner language [11]. This DSL contains operators that map another combiner across the components of a stream, select among multiple streams, combine multiple stream components into a single component, rerun the command on the concatenated input streams, and combine multiple streams into a single stream in a variety of ways.

KumQuat targets commands f with combiner g such that

$$f(\mathtt{x}_1 \mathbin{++} \mathtt{x}_2) = g(f(\mathtt{x}_1), f(\mathtt{x}_2))$$

for all input streams $\mathtt{x}_1, \mathtt{x}_2$, where the streams are structured as units separated by delimiters and terminate with newlines. ++ denotes string concatenation.

To synthesize these combiners, KumQuat repeatedly generates input streams $\mathtt{x}_1$ and $\mathtt{x}_2$, feeds them to the serial and parallel versions of f instantiated with candidate combiners g, and compares the resulting serial and parallel outputs to discard candidate combiners g that do not satisfy the equation [11]. In our example this process quickly produces the correct DSL combiners: (1) **concat** for "`cat $IN`", (2) **rerun** for "`tr -cs A-Za-z '\n'`", (3) **concat** for "`tr A-Z a-z`", (4) **merge** for "`sort`", (5) (**stitch2** ' ' **add first**) for "`uniq -c`", and (6) **merge**(`'-rn'`) for "`sort -rn`". Note that the correct combiner depends both on the command and its flags.

KumQuat then uses the synthesized combiners to parallelize the pipeline. By default, KumQuat splits the input stream before each such command and applies the combiner after the command (Figure 1b). In many cases, however, it is possible to enhance parallel performance (exploiting pipelined parallelism across consecutive commands) by eliminating intermediate combiners so that the output substreams for one command feed directly into the input substreams for the direct parallel execution of the next command (Figure 1c).

3 Experimental Results

We evaluate KumQuat on 70 benchmark scripts, including mass-transit analytics [13], natural language processing [4, 8], classic Unix one-liners [1, 2, 6, 9, 12], and Unix50 scripts [3, 7]. The benchmarks contain 133 unique command/flag combinations (referred to as "commands" below), where 121 process data and read an input stream.

KumQuat synthesizes a combiner for 113 of the 121 unique commands. No combiners are synthesized for the remaining 8, where 7 have no correct combiners. Using the synthesized combiners, KumQuat parallelizes 325 of the 427 pipeline stages (76.1%) in the benchmarks and eliminates 144 (44.3%) during optimization. The resulting parallel scripts produce the same outputs as original. Among the scripts whose serial execution time is at least 3 minutes, the unoptimized parallel speedup ranges between 3.5–14.9× (median 8.5×) and the optimized parallel speedup ranges between 3.8–26.9× (median 11.3×). These results highlight KumQuat's ability to exploit the data parallelism implicitly present in the benchmarks.

Acknowledgments. This research was funded by DARPA (HR001120C0191, N6600120C4025, and HR00112020013).

References

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

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

[3] Pawan Bhandari. 2020. *Solutions to unixgame.io.* https://git.io/Jf2dn

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

[5] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. 2021. An Order-Aware Dataflow Model for Parallel Unix Pipelines. *Proc. ACM Program. Lang.* 5, ICFP, Article 65, 28 pages.

[6] Dan Jurafsky. 2017. Unix for Poets. https://web.stanford.edu/class/cs124/lec/124-2018-UnixForPoets.pdf

[7] Nokia Bell Labs. 2019. *The Unix Game—Solve puzzles using Unix pipes.* https://unixgame.io/unix50

[8] Christopher Manning. 2016. *Unix for Poets (in 2016).* https://web.stanford.edu/class/archive/linguist/linguist278/linguist278.1172/notes/278-UnixForPoets.pdf

[9] 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.

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

[11] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 2021. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. *CoRR* abs/2012.15443 (2021). https://arxiv.org/abs/2012.15443

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

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

[14] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. 2021. PaSh: Light-Touch Data-Parallel Shell Processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 49–66.