Practically Correct, Just-in-Time Shell Script Parallelization

OSDI 2022

binpa.sh github.com/binpash/pash
Practically Correct, Just-in-Time Shell Script Parallelization

or how to get from this:

binpa.sh  github.com/binpash/pash
Practically Correct, Just-in-Time Shell Script Parallelization

or how to get from this: github.com/binpash/pash

OSDI 2022 to this:

binpa.sh  🌐  github.com/binpash/pash
Joint work with:

Tammam Mustafa  Jan Bielak  Dimitris Karnikis  Thurston Dang  Michael Greenberg  Nikos Vasilakis

Building on work by many others (in alphabetical order):

Achilles Benetopoulos  Lazar Cvetkovic  Shivam Handa  Kostas Mamouras  Radha Patel  Martin Rinard
shell
Used by everyone!

- Orchestration
  - Kubernetes deployment
  - Docket containers …

- Data processing:
  - Downloading
  - Extracting
  - Preprocessing
  - Querying

- Automation Tasks
  - Configuration
  - Installation
Used by everyone!

- Orchestration
  - Kubernetes deployment
  - Docket containers …
- Data processing:
  - Downloading
  - Extracting
  - Preprocessing
  - Querying
- Automation Tasks
  - Configuration
  - Installation

```bash
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
```
Used by everyone!

- Orchestration
  - Kubernetes deployment
  - Docket containers …

- Data processing:
  - Downloading
  - Extracting
  - Preprocessing
  - Querying

- Automation Tasks
  - Configuration
  - Installation

```bash
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: /"
do
echo "Building parser..."
eval $(opam config env)
cd compiler/parser
echo "|-- installing opam dependencies..."
make opam-dependencies
echo "|-- making libdash..."
make libdash
echo "|-- making parser..."
make
cd ../../
echo "Building runtime..."
cd runtime/ ; make ; cd ../
```
Used by everyone!

- Orchestration
  - Kubernetes deployment
  - Docket containers ...

- Data processing:
  - Downloading
  - Extracting
  - Preprocessing
  - Querying

- Automation Tasks
  - Configuration
  - Installation

```bash
# Check all possible clusters, as your .KUBECONFIG may have multiple contexts:
kubectl config view -o jsonpath='{"Cluster name":$server\{range .clusters[*]\}{.name}{"\"\"\"}{.cluster.server}{"\"\"\"}\end\'}

# Select name of cluster you want to interact with from above output:
export CLUSTER_NAME=some_cluster_name

# Point to the API server referring to the cluster name
APISERVER=$(kubectl config view -o jsonpath='{.clusters[?(@.name=="$CLUSTER_NAME")].cluster.server}')

# Gets the token value
TOKEN=$(kubectl get secrets -o jsonpath='{.items[?(@.metadata.annotations["kubernetes\.io/service-account\.name"]="default")].data.token}')[base]
for y in {2015..2019}; do
curl $base/$y | grep gz | tr -s " "
  sed "s;^;$base/$y/;" | xargs -n 1 curl -L $base/
  sed "s/^/Maximum temperature for $y is: /"
  done

# Explore the API with TOKEN

curl -X GET $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

```
echo "Pp" | eval $(opam ...)
cd compiler/parser
echo "|-- installing ..." make opam-dependencies
make libdash
echo "|-- making libdash..." make libdash
echo "|-- making parser..." make
make cd ../../
```

Used by everyone!

- Orchestration
- Kubernetes deployment
- Docket containers

Data processing:
- Downloading
- Extracting
- Preprocessing
- Querying

Automation Tasks:
- Configuration
- Installation

```bash
# Check Kubernetes deployment
PLATFORM=$(uname -m | tr '[[:upper:]]' '[[:lower:]]')
URL='http://get.pash.ndr.md/
VERSION='latest'
DOWNLOADER='curl'
alias curl='curl -s'
download () { command -v curl >/dev/null 2>&1 || { DOWNLOADER='wget'; alias curl='wget -qO-'; }
    command -v $1 >/dev/null 2>&1 && echo 'true' || echo 'false';
}
cmd_exists () {
    command -v $1 >/dev/null 2>&1 && echo 'true' || echo 'false';
}
if [ $PLATFORM = 'darwin' ]; then
echo 'pash is not yet well supported on OS X'
fi
INSTALL=0
while getopts 'i' opt do
    case $opt in
        i) INSTALL=1 ;;
    esac
esac
if [ INSTALL -eq 1 ]; then
    echo 'Error in command line parsing' >&2
    exit 1
fi
wget -O- $base
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
```
Used by everyone!

- Orchestration
- Kubernetes deployment
- Docket containers

Data processing:
- Downloading
- Extracting
- Preprocessing
- Querying

Automation Tasks
- Configuration
- Installation

```bash
for y in {2015..2019}; do
curl $base/$y | grep gz | tr -s | cut -d- -f9 | sed "s;^/; $base/$y/;" | xargs -n1 curl -s | gunzip | cut -c89-92 | grep -iv 999 | sort -rn | head -n1 | sed "s/^/Maximum temperature for $y is: /"
done
```
... for real

from the 2021 state of the octoverse: https://octoverse.github.com
Why? … well, the shell is great

- Universal Composition
  - Composing arbitrary commands using files and pipes
  - Allows users to create powerful but succinct scripts
- Unix native
  - It is well suited to the Unix abstractions (files, strings, etc)
  - Offers great control and management of the file system
- Interactive
  - The complete system environment is accessible
  - Short commands and flags allows for quick experimentation
An example: Temperature Analysis

- This script computes the max temp in the US for the years 2015-2019
- To do so it:
  - Fetches the indexes of temperature data archives
  - Downloads the archived temp data
  - Extracts the raw data
  - Cleans it
  - Computes the maximum

- The preprocessing part is taken from the Hadoop book
  - Until the gunzip

- The final two lines replace the MapReduce program from Hadoop book
  - The MapReduce equivalent in Java is 150 lines of code :')
The shell is great but ... Shell scripts are mostly sequential!*

*Actually they have a ton more issues but we will come to that in the end
The shell is great but ...  

Shell scripts are mostly sequential!*

Parallelizing requires a lot of manual effort:
- Using specific command flags (e.g., sort -p, make -jN)
- Using parallelization tools (e.g., GNU parallel)
- Rewriting script in parallel languages (e.g. Erlang)

What did we do to deserve this??? :'(  

*Actually they have a ton more issues but we will come to that in the end
PaSh
PaSh

No tight coupling: Could work on top of any shell!
PaSh on Temperature Analysis

82GB (5y weather data)

<table>
<thead>
<tr>
<th>Module</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td>bash</td>
</tr>
<tr>
<td>Processing</td>
<td></td>
</tr>
</tbody>
</table>
PaSh on Temperature Analysis

82GB (5y weather data)

Preprocessing

- bash: 33m58s
- pash -w 16: 16m39s

Processing

- 10m4s
- 49s

2.04x speedup for preprocessing

12.31x speedup for processing

2.52x combined speedup for the full program

Hadoop only focuses on this part

This part is not the focus of traditional parallelization frameworks but parallelizing it has the biggest impact.
PaSh Insights

Command Specification Framework

grep -v -f pats.txt in.txt
PaSh Insights

Command Specification Framework

```
grep -v pats in
```

Order Aware Dataflow Model

```
Parallelizing Transformations
```

Transformations proven correct

Formalized

Read the PaSh papers at EuroSys 21 and ICFP 21 for more!
PaSh -- The static way
That should be OK, right?
That should be OK, right?
Conservative or unsound – Choose one

- The shell is dynamic:
  - Current directory
  - Environment variables
  - Unexpanded strings
  - File system
Conservative or unsound – Choose one

- The shell is dynamic:
  - Current directory
  - Environment variables
  - Unexpanded strings
  - File system

- Static parallelization has to choose:
  - Sound but **conservative**
  - **Unsound** and optimistic

```bash
IN=${IN:-$TOP/pg}
mkdir $IN
cd $IN
echo 'Downloading, 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/$input" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
  sort > "$OUT/$input.out"
done
```
PaSh-JIT
Just in time parallelization

- PaSh-JIT tries to parallelize as-late-as-possible™
- Provides critical information to the compiler:
  - State of shell, Variables, Directory, Files
- Not only correct, but also faster!!!
- How?
  - By constantly switching between evaluation and parallelization
Just in time parallelization

```bash
OUT=${OUT:-$TOP/out}
for input in $(ls ${IN}); do
cat "${IN}/${input}" |
  tr -sc '[A-Z][a-z]' '[012*]' |
  sort > "${OUT}/${input}.out"
done
```

<table>
<thead>
<tr>
<th>Shell mode</th>
<th>PaSh mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP=/pash</td>
<td></td>
</tr>
<tr>
<td>IN=/pash/in</td>
<td></td>
</tr>
</tbody>
</table>
Just in time parallelization

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

Shell mode

PaSh mode

TOP=/pash
IN=/pash/in
Just in time parallelization

```
OUT=/pash/out
for input in $(ls ${IN}); do
cat "${IN}/${input}" |
  tr -sc '[A-Z][a-z]' '[\012*]' |
sort > "${OUT}/${input}.out"
done
```

```
TOP=/pash
IN=/pash/in
```
Just in time parallelization

```bash
OUT=/pash/out
for input in $(ls ${IN}); do
cat "${IN}/${input}" |
   tr -sc '[A-Z][a-z]' '[:012*]' |
sort > "${OUT}/${input}.out"
done
```

Shell mode

PaSh mode

TOP=/pash
IN=/pash/in
OUT=/pash/out
Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
cat "$IN/$input" |
   tr -sc '^[A-Z][a-z]' '[\012*]' |
sort > "${OUT}/${input}.out"
done
```

Shell mode

PaSh mode

TOP=/pash
IN=/pash/in
OUT=/pash/out
Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
  cat "$IN/$input" | tr -sc '^[A-Z][a-z]' '[012*]' | sort > "${OUT}/${input}.out"
done
```
Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
cat "$IN/$input" |
   tr -sc '[A-Z][a-z]' '[]0123' |
sort > "${OUT}/${input}.out"
done
```

Shell mode

PaSh mode

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

Expanding
Just in time parallelization

OUT=/pash/out
for input in in1 in2; do
cat "$IN/$input" |
    tr -sc '[A-Z][a-z]' '[\012*]' |
sort > "${OUT}/${input}.out"
done
Just in time parallelization

```bash
OUT=/pash/out
for input in in1 in2; do
cat /pash/in/in1 |
   tr -sc '[A-Z][a-z]' '[\012*]' |
   sort > /pash/out/in1.out
done
```

---

**Shell mode**

- `TOP=/pash`
- `IN=/pash/in`
- `OUT=/pash/out`
- `input=in1`

---

**PaSh mode**

- `TOP=/pash`
- `IN=/pash/in`
- `OUT=/pash/out`
- `input=in1`
Just in time parallelization

```sh
OUT=/pash/out
for input in in1 in2; do
cat /pash/in/in1 | tr -sc '[A-Z][a-z]' '[\012*]' | sort > /pash/out/in1.out
done
```

<table>
<thead>
<tr>
<th>Shell mode</th>
<th>PaSh mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP=/pash</td>
<td>Parallelize?</td>
</tr>
<tr>
<td>IN=/pash/in</td>
<td></td>
</tr>
<tr>
<td>OUT=/pash/out</td>
<td></td>
</tr>
<tr>
<td>input=in1</td>
<td></td>
</tr>
</tbody>
</table>
Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    mkfifo f1 f2 f3 f4
    cat /pash/in/in1 | split f1 f2 &
    ... &
    sort < f1 > f3 &
    sort < f3 > f4 &
    sort -m f3 f4 > /pash/out/in1.out
    rm f1 f2 f3 f4
done
```

Shell mode

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```

PaSh mode

```
Parallelize?
Success!
```

Success!
Just in time parallelization

```
OUT=/pash/out
for input in in1 in2; do
    mkfifo f1 f2 f3 f4
    cat /pash/in/in1 | split f1 f2 &
    ... &
    sort < f1 > f3 &
    sort < f3 > f4 &
    sort -m f3 f4 > /pash/out/in1.out
    rm f1 f2 f3 f4
done
```

Shell mode

```
TOP=/pash
IN=/pash/in
OUT=/pash/out
input=in1
```
PaSh-JIT overview

Script → Preprocessor → Parsing Library → User Shell

User Shell

Instrumented Script

... source jit.sh
...

Shell state
- Variables
- Files
- ...

JIT Engine

Compilation Server
JIT Engine
JIT Engine

Hides compilation from the perspective of the shell

<table>
<thead>
<tr>
<th>shell mode</th>
<th>PASH-JIT mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>E</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>debug mode</td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>R</td>
</tr>
</tbody>
</table>

- **S**: Save shell state and set PASH-JIT state
- **C**: Query parallelizing compiler server
- **R**: Restore shell state
- **E**: Execute (optimized or original) fragment
- **D**: Gather execution and debug information
Compilation Server

• The compilation server reduces latency!
  • Doesn’t require initializations and keeps state in memory
  • Necessary for feasibility in practice (in tight loops every ms counts)

• Also enables additional optimizations
  • Parallelization of independent fragments (e.g., iterations that touch different files)
  • Profile-guided optimizations (e.g., configuring parallelization width)

• For more, check out our paper
Evaluation
Evaluation: Correctness
Evaluation: Correctness
Evaluation: Correctness

- 1007 assertions
- 408 tests
- 29k LOC
- Covers all shell behavior
- Many edge cases

Platform: POSIX Shell Test Suite
Evaluation: POSIX test suite

• Out of the 408 tests
  • Bash passes 376 and fails 32 tests
  • PaSh-JIT passes 374 and fails 34 tests

• Divergence in these two tests is only in the exit status
  • Both return with an error, though different code
Evaluation: POSIX test suite

• Out of the 408 tests
  - Bash passes 376 and fails 32 tests
  - PaSh-JIT passes 374 and fails 34 tests
• Divergence in these two tests is only in the exit status
  - Both return with an error, though different code
• Other shells compared to bash:

<table>
<thead>
<tr>
<th></th>
<th>Both Bash and X fail</th>
<th>Bash succeeds X fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>dash</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>ksh</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>mksh</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>posh</td>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>yash</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>
Evaluation: POSIX test suite

• Out of the 408 tests
  • Bash passes 376 and fails 32 tests
  • PaSh-JIT passes 374 and fails 34 tests

• Divergence in these two tests is only in the exit status
  • Both return with an error, though different code

• Other shells compared to bash:

• Various shell failures on POSIX tests:

<table>
<thead>
<tr>
<th>Shell</th>
<th>Both Bash and X fail</th>
<th>Bash succeeds X fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>dash</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>ksh</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>mksh</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>posh</td>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>yash</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>
Evaluation: POSIX test suite

- Out of the 408 tests
  - Bash passes 376 and fails 32 tests
  - PaSh-JIT passes 374 and fails 34 tests
- Divergence in these two tests is only in the exit status
  - Both return with an error, though different code
- Other shells compared to bash:
- Various shell failures on POSIX tests:

<table>
<thead>
<tr>
<th>Shell</th>
<th>Both Bash and X fail</th>
<th>Bash succeeds X fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>dash</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>ksh</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>mksh</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>posh</td>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>yash</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

By following a lightweight shim approach (instead of reimplementing) we achieve very high compatibility with bash 🎉
Evaluation: Performance

- Evaluating on 82 shell scripts (4 suites and 11 standalone scripts)

Avg speedups: PaSh-JIT (x5.8) – PaSh-AOT (x2.9)
Conclusion
Conclusion

• Shells were angry that we tried to parallelize statically
Conclusion

• Shells were angry that we tried to parallelize statically
• We can make them happy by being dynamic
Conclusion

• Shells were angry that we tried to parallelize statically
• We can make them happy by being dynamic
• Correct
• And fast!
The shell has more problems...

- Error-proneness
  - accidentally `rm -rf /`

- Hard to learn
  - still googling for if-then-else shell syntax

- Redundant recomputation
  - we have to use Makefiles etc

- Lack of support for contemporary deployments
  - managing a distributed cluster

Recent exceptions: Rattle [1] and Riker [2]


The JIT part of PaSh-JIT is an enabler

- The JIT structure of PaSh-JIT enables additional analyses/solutions
Some exciting future directions

• A shell monitor that ensures that safety/security props are not violated
• A fully distributed shell 🧵
• An incremental execution shell
• Talk to us if you have ideas!
• Michael and Nikos are hiring!
Practical impact and availability

• PaSh is open source and hosted by the Linux Foundation

• It is virtually indistinguishable from bash (406/408 POSIX tests)
  • And requires no modifications/reimplementation

• OSDI artifact badges

• Download it and play [binpa.sh](https://github.com/binpash/pash)