Fast regression testing of TEX packages: Multiprocessing and batching

Vít Starý Novotný, Marei Peischl

Abstract

In the version 3.0.0 of the Markdown package for TEX, the number of regression tests increased from 143 to 783. This caused the tests to run for up to 15 hours, which slowed down our development cycle. In this article, we describe a novel technique for batching test files that reduced our testing time from 15 hours to just 15 minutes. With batching, the amount of time that is spent on actual testing increased from 5% to 97%. When combined with multiprocessing on 32 CPUs, our batching technique achieved a speed increase of up to 161 times compared to running without any multiprocessing or batching.

1 Introduction

Small TEX packages, typically developed in a single iteration rather than through ongoing updates, can depend on user feedback to maintain the code. However, this approach has its limitations. Larger projects, especially those that are continuously developed, require a more robust solution. Automated regression tests are crucial in these cases. They ensure that any changes, either in the code itself or its external dependencies, do not alter the expected behavior of the code.

The Markdown package for TEX also features a set of regression tests. These tests, designed to be completed in just a few minutes, provide immediate feedback and are automatically conducted on any updates submitted to the package's GitHub repository.

After the implementation of the CommonMark standard in version 3.0.0 of the Markdown package, the number of tests increased from 143 to 783 (about a 5.5-fold increase). This caused the tests to take up to 15 hours to run, using free GitHub-hosted runners, too slow to provide any benefit to developers.

In order to increase the testing speed, we implemented a novel technique for batching test files and we added self-hosted runners with up to 12 CPUs. After these changes, the tests finish in about 15 minutes, which is a 60-fold speed increase and which makes the tests practically useful to developers.

In this article, we describe the testing framework of the Markdown package. In sections 2 through 4, we describe the definition files, techniques, and strategies used in our framework. In Section 5, we describe the details of our implementation. In sections 6 and 7, we describe our experiments and their results. In Section 8, we discuss prior work related to our framework. We conclude in Section 9 by summarizing our contributions and outlining future work.

2 Definition files

In the future, an AI agent might examine the code of a TEX package and identify any incorrect behavior. For the moment, regression testing requires the manual creation of many definition files that describe the expected behavior and how it should be validated.

In this section, we describe the definition files used in our framework: test files, formats, commands, and templates.

2.1 Test files

The Markdown package converts markdown text to TEX commands. To validate the conversion, our framework redefines these TEX commands to produce output in the `.log` file, which we then examine.

A *test file* consists of a) TEX code that configures the Markdown package, b) markdown text, and c) the expected output in the `.log` file.

As an example, `strike-through.test` tests the strike-through syntax extension:

```
\markdownSetup
  { strikeThrough }      a
<<<
Hello ~~world~~!       b
>>>
BEGIN document
strikeThrough: world     c
END document
```

strike-through.test

2.2 Formats, commands, and templates

The Markdown package supports several combinations of TEX formats and engines. For each TEX format, there are also several ways to input markdown text. Our framework ensures that a markdown text always produces the same output.

A *format* consists of one or more a) *commands* that can be used to typeset documents in a TEX format using different TEX engines and b) *templates* that specify the different ways in which markdown text can be input with the TEX format.

An example format `plain` contains commands for the pdfTEX, XƎTEX, and LuaTEX engines:

```
pdftex --shell-escape
↪  TEST_FILENAME
xetex --shell-escape
↪  TEST_FILENAME
luatex TEST_FILENAME[1]
```

COMMANDS.m4

[1] The Markdown package uses Lua to parse markdown text. Whereas LuaTEX can execute Lua code directly, other TEX engines must use the shell of the operating system to

The format plain also contains two templates. One uses the \markdownInput TeX macro and the other one uses the \markdownBegin and End TeX macros:

```
\input markdown
\input TEST_SETUP_FILENAME
\markdownInput
↪  {TEST_INPUT_FILENAME}
\bye
```

input.tex.m4

```
\input markdown
\input TEST_SETUP_FILENAME
\markdownBegin
undivert(TEST_INPUT_FILENAME)
\markdownEnd
\bye
```

verbatim.tex.m4

2.3 Materialized templates and commands

During testing, the texts TEST_SETUP_FILENAME and TEST_INPUT_FILENAME in templates are replaced with names of auxiliary files that contain the TeX code and the markdown text parts of a test file, respectively. Also, the text undivert(TEST_INPUT_FILENAME) is replaced with the literal markdown text from the test file. After the replacement, we say that the template has been *materialized*.

Here is the template verbatim.tex.m4 from Section 2.2 after it has been materialized with the test file strike-through.test from Section 2.1:

verbatim.tex.m4 + strike-through.test ⤵

```
\input markdown
\input test-setup.tex
\markdownBegin
Hello ~~world~~!
\markdownEnd
\bye
```

verbatim.tex

```
\markdownSetup
  { strikeThrough }
```

test-setup.tex

After a template has been materialized, the text TEST_FILENAME in all commands is replaced with the filename of the materialized template. After the replacement, the command has also been materialized.

execute Lua code. Since accessing the shell is a security risk, users must express their consent by writing --shell-escape.

Here are the commands COMMANDS.m4 from Section 2.2 after they have been materialized:

COMMANDS.m4 + verbatim.tex ⤵

```
pdftex --shell-escape
↪  verbatim.tex
xetex --shell-escape
↪  verbatim.tex
luatex verbatim.tex
```

COMMANDS

During testing, the materialized commands are executed. Each command produces a .log file, which is compared to the expected output from the test file.

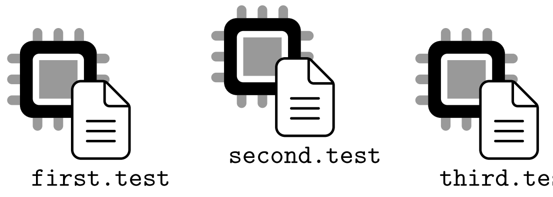
3 Computational techniques

While testing all combinations of test files, templates, and commands ensures comprehensive coverage of all potential configurations, it can be time-consuming.

In this section, we describe the computational techniques of multiprocessing, the batching of test files, and how they increase the speed of testing in our framework. Furthermore, the batching of test files raises challenges with load balancing and the attribution of errors. We discuss the challenges and describe the techniques of batch size limiting and batch splitting to address them.

3.1 Multiprocessing

Whereas TeX uses only a single CPU, modern PCs can contain several CPUs. Therefore, we can increase the speed of testing by using *multiprocessing*, where each CPU processes a different test file:



Using N CPUs increases testing speed up to N times.

3.2 Batching of test files

At the beginning of a document, TeX initializes packages, fonts, Lua scripts, and other assets, which slows down testing:

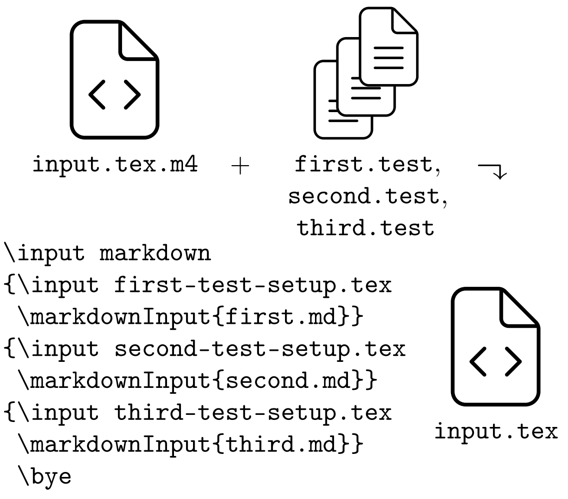
```
\input markdown
\input test-setup.tex
\markdownBegin
Hello ~~world~~!
\markdownEnd
\bye
```

slow

fast

verbatim.tex

To increase the speed of testing, we can amortize the cost of initialization by materializing a template with a *batch* of several test files:

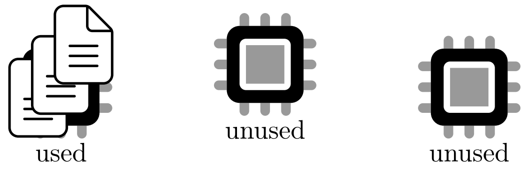


```
\input markdown
{\input first-test-setup.tex
 \markdownInput{first.md}}
{\input second-test-setup.tex
 \markdownInput{second.md}}
{\input third-test-setup.tex
 \markdownInput{third.md}}
 \bye
```

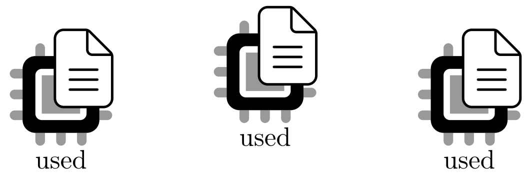
Batching N test files decreases initialization cost N times. How much this speeds up testing depends on the ratio between the time spent on initialization and the time spent on processing the rest of the template.

3.3 Batch size limiting

When we use both multiprocessing and batching with large batch sizes, most CPUs will be unused:



We can *limit the batch size* so that all CPUs are used:



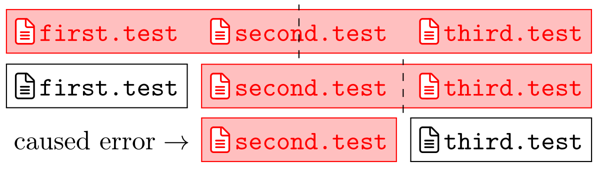
Limiting the batch size increases the speed of testing, because every used CPU has less work to do.[2]

3.4 Batch splitting

When we test batches of test files, a `.log` file is split into sections corresponding to individual test files and compared with the expected test file outputs. However, if a fatal error occurs, the `.log` file may become malformed. To find the test file responsible for the error, we repeatedly *split the batch* using binary search.

[2] However, the CPUs do more work overall, because every used CPU has to pay the initialization cost and more CPUs are used. Therefore, limiting the batch size increases the speed of testing but decreases energy efficiency.

Here is how we would split a batch of test files `first.test`, `second.test`, and `third.test`, where `second.test` causes a fatal error:



First, we try processing all files together but we encounter a fatal error. Therefore, we divide the files into two groups: one with `first.test` and the other with `second.test` and `third.test`. Processing these separately, we again face a fatal error in the group with `second.test` and `third.test`. We then split this group into two individual files, `second.test` and `third.test`, and we process them. The fatal error occurs with `second.test`, which we identify as the cause of the error.

When only one test file out of N files in a batch causes a fatal error, batch splitting executes at most $2(\log_2 N+1)$ commands. This is less than or equal to N for sufficiently large batch sizes $N \geq 8$. Therefore, in the presence of no more than a few fatal errors, batching is still faster than sequential processing.

4 Error handling strategies

Developers and maintainers have different needs when it comes to the handling of errors. Whereas developers need immediate feedback during the development of new features, maintainers require a comprehensive summary of all errors when they deal with unexpected breakage.

In this section, we describe the error handling strategies of developer- and maintainer-oriented testing and updating test files. We also discuss how these strategies increase the speed of development and decrease maintenance costs.

4.1 Developer-oriented testing

In the practice of test-driven development, before adding a new feature, developers first write new test files that describe the expected behavior of the feature. Then, they develop the feature until all test files have passed. At the beginning, old test files will pass, whereas new test files will fail. At the end, all test files, both old and new, should pass.

In order to increase the speed of development, tests should fail fast to provide immediate feedback. Therefore, developers can configure our framework to start with the new test files, which are the most likely to fail, and stop at the first error rather than wait until all tests have finished.

For example, imagine a TEX package with two test files: `first.test` and `second.test`. For simplicity, the package has only one format with one template and with three commands for the pdfTEX, XƎTEX, and LuaTEX engines. Before the development of a new feature, developers add a new test file `third.test` and they run the tests with the following results:

	pdfTEX	XƎTEX	LuaTEX
`first.test`			
`second.test`			
`third.test`	✗		

Since `third.test` was new, it was tested first and immediately failed, providing immediate feedback to developers. This is how we test all updates submitted to the GitHub repository of the Markdown package.

4.2 Maintainer-oriented testing

Tests can fail not just during the development of new features but also during maintenance. These errors are often caused by changes to external dependencies such as TEX engines, formats, and packages.

In order to decrease maintenance costs, tests should provide comprehensive feedback. Therefore, maintainers can configure our framework to always process all test files and produce a helpful summary of all errors.

Continuing the example from the previous section, developers finish the new feature and release an updated version of their package on CTAN. However, after a month, the tests fail with the following results:

	pdfTEX	XƎTEX	LuaTEX
`first.test`	✓	✗	✓
`second.test`	✓	✗	✓
`third.test`	✓	✗	✓

At a glance, the summary shows that the errors are related to the XƎTEX engine. This is how we test the Markdown package every week.

4.3 Updating test files

Although test-driven development is well-suited to adding new features, fundamental changes to the code may require that existing test files are updated as well. Furthermore, writing test files before the development of a feature can be difficult, especially for complex features with incomplete requirements.

In order to increase the speed of development, developers can configure our framework to *update test files* instead of failing. Developers can make fundamental changes and our framework will update the expected outputs in test files to match the actual output. Furthermore, developers can also develop a feature, write partial test files for the feature that contain only the TEX code and markdown text, and use our framework to fill in the expected output. Then, developers can review the changes and determine whether they are correct.

Our framework will update a test file only if all templates and commands produce consistent outputs. In the example from the previous section, the command for the XƎTEX engine failed for all test files, whereas the other commands did not. Therefore, our framework would not update any test files and fail.

5 Implementation

Before Markdown 3.0.0, our framework was implemented by a Bash script `test.sh`; see Listing 1.

At first, `test.sh` processed test files sequentially and did not use the computational techniques from Section 3 to increase the speed of testing. Since Markdown 2.4.0, we used the GNU Parallel command-line tool [7] to implement multiprocessing:

```
$ find -name '*.test' | parallel ./test.sh
```

Out of the error handling strategies from Section 4, `test.sh` could only update test files.

While Bash is convenient for simple programs, more complicated programs are better written in a more expressive language. In Markdown 3.0.0, we rewrote our framework from Bash to Python 3 [4].

Out of the techniques and strategies from Sections 3 and 4, the higher expressiveness of Python allowed us to implement the batching of test files, developer- and maintainer-oriented testing, and batch splitting. Furthermore, Python's built-in support for multiprocessing allowed us to stop using GNU Parallel and implement batch size limiting.

test.sh

```
#!/bin/bash
set -o errexit -o pipefail -o nounset
BUILDDIR="$(mktemp -d)"
trap 'rm -rf "$BUILDDIR"' INT TERM
for TESTFILE; do
  printf 'Testfile %s\n' "$TESTFILE"
  for FORMAT in templates/*/; do
    printf '  Format %s\n' "$FORMAT"
    for TEMPLATE in "${FORMAT}"*.tex.m4; do
      printf '    Template %s\n' "$TEMPLATE"
      m4 -DTEST_FILENAME=test.tex <"$FORMAT"/COMMANDS.m4 |
      (while read -r COMMAND; do
        printf '      Command %s\n' "$COMMAND"

        # Set up the testing directory.
        cp support/* "$TESTFILE" "$BUILDDIR"
        cd "$BUILDDIR"
        sed -r '/^\s*<<<\s*$/{x;q}' \
          <"${TESTFILE##*/}" >test-setup.tex
        sed -rn '/^\s*<<<\s*$/,/^\s*>>>\s*$/{/^\s*(<<<|>>>)\s*$/!p}' \
          <"${TESTFILE##*/}" >test-input.md
        sed -n '/^\s*>>>\s*$/,${/^\s*>>>\s*$/!p}' \   a
          <"${TESTFILE##*/}" >test-expected.log
        m4 -DTEST_SETUP_FILENAME=test-setup.tex \
           -DTEST_INPUT_FILENAME=test-input.md <"$OLDPWD"/"$TEMPLATE" >test.tex

        # Run the test, filter the output and concatenate adjacent lines.
        eval "$COMMAND" >/dev/null 2>&1 ||
          printf '        Command terminated with exit code %d.\n' $?    b
        touch test.log
        sed -nr '/^\s*TEST INPUT BEGIN\s*$/,/^\s*TEST INPUT END\s*$/{
          /^\s*TEST INPUT (BEGIN|END)\s*$/!H
          /^\s*TEST INPUT END\s*$/{s/.*//;x;s/\n//g;p}
        }' <test.log >test-actual.log

        # Compare the expected outcome against the actual outcome.
        diff -a -c test-expected.log test-actual.log ||           c
        # Uncomment the below lines to update the testfile.
#          (sed -n '1,/^\s*>>>\s*$/p' <"${TESTFILE##*/}" &&
#           cat test-actual.log) >"$OLDPWD"/"$TESTFILE" ||    d
           false

        # Clean up the testing directory.
        cd "$OLDPWD"
        find "$BUILDDIR" -mindepth 1 -exec rm -rf {} +
      done)
    done
  done
done
rm -rf "$BUILDDIR"
```

Listing 1: The shell script `test.sh` that implemented the testing framework of the Markdown package before version 3.0.0. For each test file, `test.sh` a) materializes templates in a temporary directory, b) executes materialized commands, c) compares the `.log` file against the expected output, and d) optionally updates the test file.

6 Experiments

In this section, we describe our experiments with multiprocessing and batching. In our experiments, we aimed to answer the following research questions:

1. What is the speed benefit of multiprocessing?
2. What is the speed benefit of batching test files?
3. What is the speed benefit of batch size limiting?

To answer the questions, we tested the Markdown package with different CPU counts and batch sizes:

- Numbers of CPUs: 1, 2, 4, 8, 16, and 32
- Batch sizes: 1, 2, 4, 8, ..., 256, 512, and 1024

To ensure reliability of our findings, we repeated each test configuration five times and we measured the median testing time to control for sample variance. Our experimental code is available online. [6]

We tested the Markdown package at Git commit `7613632` from August 21, 2023. At this commit, the Markdown package contained 783 test files. For each test file, 14 commands were materialized and executed: four for plain TeX, four for LaTeX, and six for ConTeXt MkIV. Therefore, TeX formats were initialized up to $14 \cdot 783 = 10{,}962$ times during testing.

To show the speed benefit of batch size limiting, we deactivated it in our experiments, thereby highlighting the speed reduction caused by its absence.

We ran the experiments for 33 days on a shared GNU/Linux server with 400 GB of RAM and 80 CPUs, each at 2.1 GHz.

7 Results

Figure 1 shows the results of our experiments. In this section, we discuss the results and how they relate to the three research questions outlined in the previous section.

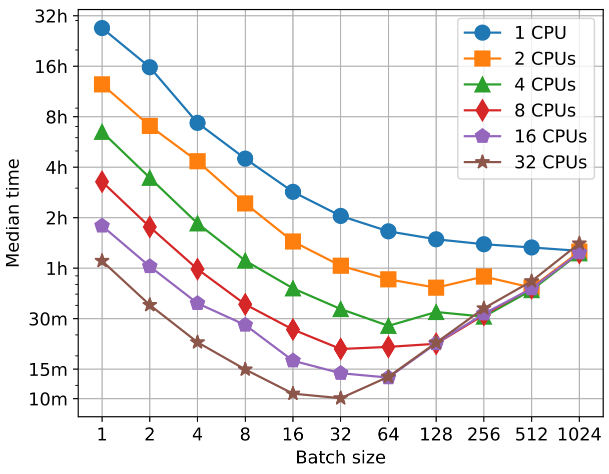


Figure 1: The median testing times for different numbers of CPUs and batch sizes

7.1 Multiprocessing

With batch size 1, the testing speed scales almost linearly with the number of CPUs, as we would expect: Whereas with 1 CPU, the median testing time is 27 hours and 2 minutes, it is only 1 hour and 6 minutes with 32 CPUs (about 24-fold speed-up).[3]

7.2 Batching of test files

With 1 CPU, the testing speed also scales almost linearly with the batch size, up to a point. Whereas with batch size 1, the median testing time is 27 hours and 2 minutes, it is only 4 hours and 30 minutes with batch size 8 (about 6-fold speed-up), and 1 hour and 20 minutes with batch size 512 (about 21-fold speed-up). This indicates that initialization dominates the testing time.

To better understand the relationship between the initialization and the testing time, we can solve the following series of equations:

$$14 \cdot (783 \cdot (X + Y)) = 27 \text{ hours and 2 minutes}$$
$$14 \cdot (X + 783 \cdot Y) = 1 \text{ hour and 16 minutes}$$

On the left-hand side of the equations, the variable X stands for the mean time that it takes to initialize a TeX format and the variable Y stands for the mean time that it takes to process the markdown text from a single test file. On the right-hand side of the equations are the median testing times with 1 CPU and batch sizes 1 (above) and 1024 (below).

The solution shows that whereas it takes a full $X \approx 8.47$ seconds to initialize a TeX format, it takes only $Y \approx 0.41$ seconds to process a markdown text. In other words, without batching, 95% of time is spent on initialization and only 5% on actual testing; with batching, up to 97% of time is spent on testing.

7.3 Batch size limiting

The speed improvements from multiprocessing and batching are additive, up to a point. Whereas with 1 CPU and batch size 1, the median testing time is 27 hours and 2 minutes, it is only 10 minutes with 32 CPUs and batch size 32 (about 161-fold speed-up).

When the number of CPUs multiplied by the batch size exceeds the number of test files (783), we cannot use all CPUs and the testing speed decreases. Whereas with 32 CPUs and batch size 16, the median testing time is only 11 minutes, it is 1 hour and 24 minutes with the same number of CPUs and batch size 1024 (about 8-fold slow-down). Our framework prevents this effect by limiting the batch size.

[3] The reason that we did not achieve the theoretical 32-fold speedup is likely tasks from other users on our server.

8 Related work

Besides our framework, there exist other frameworks for regression testing of TEX packages. Furthermore, the computational techniques in our framework are often adapted from previous work in other fields.

In this section, we discuss the regression testing framework of the l3build package management system and we compare it with our framework. Furthermore, we discuss the origin of the batching of test files and batch splitting.

8.1 The l3build package

The l3build package [2, 9, 8, 1] provides a comprehensive system for TEX package management that also includes a regression testing framework.

Whereas our framework is written in Python, l3build is written in Lua. Each language presents its own set of strengths and weaknesses. On one hand, every modern installation of TEX includes a Lua interpreter, which makes l3build more accessible for TEX users compared to our framework. On the other hand, unlike Python, Lua has no built-in support for multiprocessing. Therefore, l3build users must use external tools like GNU Parallel to use multiple CPUs for testing, whereas our framework can use multiple CPUs out of the box.

In our framework, each test file is designed to hold a single self-contained test that avoids modifying the global state. This design allows for straightforward grouping of tests into batches, where each test is isolated from others using TEX groups, as we discussed in Section 3.2.

In l3build, a single test file may contain multiple tests. These tests can be interdependent, creating a challenge in separating them from their files. Additionally, these tests might change the global state, which poses a risk of unexpected conflicts when tests are grouped into batches. Due to these complexities, l3build does not support the batching of tests. Nonetheless, the practice of including multiple tests in a single test file can be seen as a form of manual batching that amortizes the cost of initialization.

Both our framework and l3build support the updating of test files [1, Section 2.7]. This allows developers to automatically generate parts of test files when they make fundamental changes to the code or when they develop complex new features, as discussed in Section 4.3.

8.2 Batching and batch splitting

The techniques of batching and batch splitting were perhaps first used with TEX in the ARQMath competitions in large-scale indexing of math formulae.

In the first ARQMath competition, the MIRMU team used the LATEXML tool to convert TEX formulae to XML [5, Section 2.2]. Due to speed issues when processing each formula separately, MIRMU processed them in batches. However, a single error would cause the loss of an entire batch. Therefore, MIRMU used batch splitting to recover correct formulae after an error [3].

In the third ARQMath competition, the organizers used the same techniques to provide math formulae in the XML format to all participants.

9 Conclusion

Larger TEX packages commonly use regression tests to ensure code integrity over time. In this study, we explored techniques for speeding up the regression testing of TEX packages. We have shown that batching test files can improve the testing efficiency from 5% to 97%. We have also shown that multiprocessing on 32 CPUs combined with the batching of test files increases testing speed up to 161 times.

The lessons learned from our work are as follows:

- Whereas TEX uses only a single CPU, modern PCs can contain several CPUs. Multiprocessing can increase testing speed by using these CPUs.
- Writing small isolated test files is convenient for authors but carries a high initialization cost during testing. For TEX packages such as the Markdown package, where tests are easy to isolate, the batching of test files can be used to recover the initialization cost.
- Whereas developers need tests to fail as fast as possible, maintainers benefit from a comprehensive summary of all errors.

We hope that our practical lessons will improve the regression testing practices used in the development and maintenance of TEX packages. With fast regression testing, developers can quickly introduce new features, while maintainers can proactively address emerging issues before they affect users.

Disclaimer

No wolves were harmed in the making of this article.

Acknowledgements

We would like to gratefully acknowledge the Natural Language Processing Center at the Faculty of Informatics, Masaryk University, Brno, who kindly provided computational resources for our experiments.

We also wish to extend our special thanks to Joseph Wright, Ben Frank, Barbara Beeton, and Karl Berry for their invaluable insights and thorough review of our work. Their expertise and thoughtful feedback have been instrumental in shaping the final manuscript.

Donation request

Despite our breakthroughs in testing speed, additional computational resources would greatly accelerate the development and maintenance of the Markdown package for TEX.

We graciously invite donations of GitHub self-hosted runners, particularly those hosted on GNU/Linux servers with at least 16 GB of RAM and 12 CPUs. For more information, please contact us by email. Donors will be acknowledged in the project documentation, with the option to remain anonymous upon request.

References

[1] LATEX project team. l3build: A testing and building system for (LA)TEX, Nov. 2023. `ctan.org/pkg/l3build`

[2] F. Mittelbach, W. Robertson, LATEX3 team. l3build: A modern Lua test suite for TEX programming. *TUGboat* 35(3):287–293, 2014. `tug.org/TUGboat/tb35-3/tb111mitt-l3build.pdf`

[3] V. Novotný. ARQMath data preprocessing, June 2020. `github.com/MIR-MU/ARQMath-data-preprocessing/blob/main/scripts/latex_tsv_to_cmml_and_pmml_tsv.py`

[4] V. Novotný. Implement batching and summarization to unit tests, Jan. 2023. `github.com/witiko/markdown/issues/245`

[5] V. Novotný, P. Sojka, et al. Three is better than one. In *CEUR Workshop Proceedings: ARQMath task at CLEF conference*, vol. 2696, pp. 1–30, Thessaloniki, Greece, 2020. CEUR-WS. `ceur-ws.org/Vol-2696/paper_235.pdf`

[6] V. Starý Novotný. Measure the speed of tests with different numbers of processes and batch sizes, Oct. 2023. `github.com/Witiko/markdown/blob/main/experiments/2023-10-12-test-batching`

[7] O. Tange. GNU Parallel: The command-line power tool. *USENIX Mag*, 36(1):42–47, 2011. Available from `doi.org/10.5281/zenodo.8278274`.

[8] J. Wright. l3build: The beginner's guide. *TUGboat* 43(1):40–43, 2022. `tug.org/TUGboat/tb43-1/tb133wright-l3build.pdf`

[9] J. Wright, LATEX3 team. Automating LATEX(3) testing. *TUGboat* 36(3):234–236, 2015. `tug.org/TUGboat/tb36-3/tb114wright.pdf`

◇ Vít Starý Novotný
Studená 453/15
Brno 63800, Czech Republic
`witiko (at) mail dot muni dot cz`
`github.com/witiko`

◇ Marei Peischl
Gneisenaustr. 18
Hamburg 20253, Germany
`marei (at) peitex dot de`
`peitex.de`