Parallelizing Quickref

Didier Verna
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

Quickref is a global documentation project for Common Lisp software. It builds a website containing reference manuals for Quicklisp libraries. Each library is first compiled, loaded, and introspected. From the collected information, a Texinfo file is generated, which is then processed into Html. Because of the large number of libraries in Quicklisp, doing this sequentially may require several hours of processing. We report on our experiments parallelizing Quickref. Experimental data on the morphology of Quicklisp libraries has been collected. Based on this data, we are able to propose a number of parallelization schemes that reduce the total processing time by a factor of 3.8 to 4.5, depending on the exact situation.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Software and its engineering** → **Software performance**; Software libraries and repositories; • **Applied computing** → Hypertext languages;

KEYWORDS

Parallelization, Multi-Threading, Software Performance, Software Documentation, Typesetting

ACM Reference Format:
Didier Verna. 2019. Parallelizing Quickref. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2632534

1 INTRODUCTION

Quickref is a global documentation project for Common Lisp [9] software. It builds a website containing reference manuals for libraries available in Quicklisp[1]. Quickref is freely available[2], so anyone can create a local version of the documentation website for personal use, but we also maintain a public website documenting the whole Quicklisp world[3].

Quickref itself is actually not much more than a layer of integration "glue" cadencing the inter-operation of four external software components (see Section 2). Until recently, it essentially consisted in a big loop, iterating over every Quicklisp library, and sequentially executing all the steps required in producing the corresponding reference manual, from downloading the library to actually producing an Html file. Because Quicklisp is quite large (it currently provides more than 1700 libraries), this process could take between 1h30 and 7 hours on our test machine, depending on the exact conditions. Even if 7 hours, the worst case scenario, still fits nicely into one night of batch processing, it *is* worth trying to improve the performance of the system, notably by means of parallelism. The purpose of this article is to report on this work.

Section 2 describes the tool-chain involved in the generation of the reference manuals and some important characteristics of the involved software components. Section 3 presents the experimental conditions and the various configurations used to perform timing measurements. Section 4 provides and analyzes some preliminary global measurements, giving us a general idea of what to expect. Section 5 proposes different parallel solutions, each one coming with its *pros* and *cons*. Finally, after the conclusion, an extensive discussion is proposed in Section 7.

2 QUICKREF TOOL-CHAIN

Figure 1 depicts the typical reference manual production pipeline used by Quickref, for a library named `foo`.

(1) Quicklisp is first used to make sure the library is installed upfront. This is done by calling `ql-dist:ensure-installed`, and results in the presence of a directory for that library (a *release* in Quicklisp terms) in the Quicklisp directory tree. Currently, Quickref only considers one system per library, called the *primary system*. How exactly this system is computed is unimportant for this paper.
(2) Declt[4] is then run on the primary system to generate the documentation. Declt is another library of ours, written 5 years before Quickref, but with that kind of application in mind right from the start. In particular, it is for that reason that the documentation generated by Declt is in an intermediate format called Texinfo.
(3) The Texinfo file is finally processed into Html. Texinfo[5] is the GNU official documentation format. There are two main reasons why this format was chosen when Declt was originally written. First, it is particularly well suited to technical documentation. More importantly, it is designed as an abstract, intermediate format from which human-readable documentation can in turn be generated in many different forms (Pdf and Html notably).

Quickref essentially runs this pipeline on every available library (it currently has the ability to limit itself to what is already installed,

[1] https://www.quicklisp.org
[2] https://gitlab.common-lisp.net/quickref
[3] https://quickref.common-lisp.net

[4] https://www.lrde.epita.fr/~didier/software/lisp/misc.php#declt
[5] https://www.gnu.org/software/texinfo/

ELS'19, April 01–02 2019, Genova, Italy

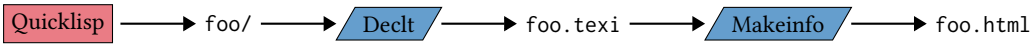

Figure 1: Reference Manual Generation
Main thread, External Process

or process the whole Quicklisp world). Some important remarks need to be made about this process.

First of all, Declt works by introspection: it uses ASDF[6]'s `load-system` function to load the system being processed, which may involve compiling and loading some dependencies as well as the system itself. It then introspects the system to collect documentation items, notably by way of the `sb-introspect` facility from SBCL[7]. Given the size of Quicklisp, it would be unreasonable to load almost two thousand libraries in a single Lisp image. For that reason, Quickref doesn't actually run Declt directly, but instead uses `uiop:run-program` to fork an SBCL script to do it.

Similarly, `makeinfo` (`texi2any` in fact), the program used to convert the Texinfo files to HTML, is an external program written in Perl (with some parts in C), not a Lisp library. Thus, here again, `uiop:run-program` is used to fork a `makeinfo` process out of the Quickref Lisp instance.

In light of these remarks, the reader must keep in mind the following points.

- Declt and Texinfo are treated as monolithic black boxes (in fact, ASDF as well), that is, we don't attempt to alter their operation. Any attempt at parallelization will hence boil down to scheduling the `declt` and `makeinfo` processes in a specific way. Thus, when we speak of "threads" in the remainder of this paper, we actually mean small Lisp functions that essentially fork external processes and wait for them, in a loop.
- Because running Declt may require the compilation of Lisp components, possibly including dependencies, and because different libraries may share the same dependencies, there is an inherent concurrency problem in writing the compilation files. Care must hence be taken to protect against those potentially concurrent accesses when needed.

3 EXPERIMENTAL CONDITIONS

3.1 Environment

All benchmarks reported in this paper were collected on a Debian Gnu/Linux[8] system, version 9.6 "Stretch". Quicklisp currently requires Debian 9 for testing, and advertises the list of required foreign dependencies. This environment hence guarantees that as many libraries as possible could be handled. All foreign dependencies were pre-installed, most of them already packaged by Debian. We used SBCL 1.4.0, cloned from its Git repository and manually compiled with `--fancy` (which, among other things, activates multi-threading).

The computer used was a Dell Precision T1600, equipped with 16 GB of RAM, a 120 GB mechanical hard drive and an Intel Xeon E3-1245 processor. This processor has 4 hyper-threaded cores[7], so that 8 threads are actually available. Note that as of version 2.4, the Linux kernel is aware of hyper-threading. Debian 9 includes version 4.9. Although the various timings reported in this paper were collected from single runs (as opposed to averaging several ones), the machine was freshly rebooted in single-user mode, in order to avoid non-deterministic operating system or hardware side-effects as much as possible.

For the Quickref tool-chain, the following software components were used: Declt 2.4.1, Makeinfo (`texi2any`) 6.5, and an up-to-date Quicklisp version 2019-01-07. This version of Quicklisp contains 1719 libraries. It is worth noting that Quickref currently fails on less than 2% of those libraries, for various reasons: dependency problems (foreign or not), compilation problems, Declt problems *etc.* These issues are out of the scope of this paper, so we simply filtered out the problematic libraries in our reports.

3.2 Configuration

While Quickref is primarily meant to build a complete documentation website for Quicklisp, a number of options are available, which need to be taken into account in our experiments.

3.2.1 Libraries and updating policy. By default, Quickref attempts to globally update Quicklisp before processing, which is the right thing to do for the public website. Individual users, however, also have the possibility to create a local website for their personal working environment only. To this end, Quickref makes it possible to only consider the libraries already installed on the local machine (instead of the whole Quicklisp world), and also to avoid updating those if that is unwanted. As a consequence, and depending on the exact situation, documenting a library with Quickref may or may not lead to downloading some code, and may or may not trigger some Lisp compilation (dependencies included) before actually loading and introspecting it.

3.2.2 Cache policy. On several occasions, we observed problems related to the compilation of common dependencies. One typical problem is when two libraries depend on a third one, and that dependency needs to be compiled in two different ways. A *global* compilation cache, as provided by ASDF by default is bound to fail. Another problem (which, at least in our opinion, should be regarded as a bug) is when the compilation or loading of a library leads to global side-effects on the top-level environment. The latest example we saw is that of `common-lisp-stat`, globally changing the reader's default float format from `single-float` to `double-float`[10]. This kind of behavior is bound to cause problems, especially when almost two thousand libraries are involved. Because of that, Quickref now has an option for making ASDF use a different, *local*, compilation cache for every documented library.

3.2.3 Scenarios. In order to take into account all those different options, we have experimented on 3 different situations.

[6] https://common-lisp.net/project/asdf/
[7] http://www.sbcl.org/
[8] http://www.debian.org

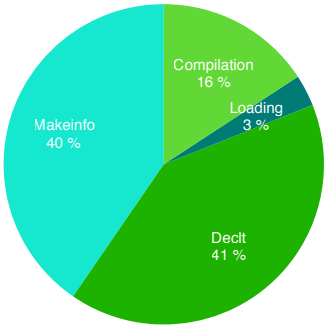

Figure 2: Time Distribution w/ compilation

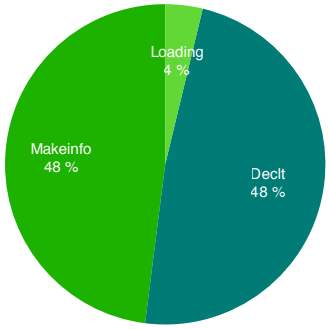

Figure 3: Time Distribution w/o compilation

(1) All the libraries are already compiled, so Declt just needs to load them. This is the best-case scenario. Also, note that whether the compilation cache is global or local doesn't matter here.
(2) The libraries are not compiled, but a global compilation cache is used, so that no redundant processing occurs. This should be regarded as the intermediate, most frequent scenario.
(3) The libraries are not compiled, and local compilation caches are used. This is the absolute worst-case scenario.

Note that regardless of the scenario, we always process the entirety of Quicklisp (modulo the failures), and the 1719 libraries in question are already downloaded. Network connectivity is considered too fluctuating to be included in benchmarks, and besides, including it would hinder the idea of experimenting in single-user mode, in order to be as deterministic as possible. Under those conditions, we measured that in the original, sequential version of Quickref, scenario 1 takes 1h 27m, scenario 2 takes 1h 51m, and scenario 3 takes 7h 01m.

4 PRELIMINARY ANALYSIS

In order to get a general idea on the behavior of the different software components involved, we performed a set of preliminary measurements, which we partially report and analyze in this section. We separately collected ASDF load/compile times, and Declt and Makeinfo processing times for every Quicklisp library. As of this writing, the code used to collect that experimental data is available in the `benchmark` branch of the Quickref repository. The data itself is also publicly available[9].

4.1 Time Distribution

Figures 2 and 3 depict the time distribution for scenarios 2 and 1 respectively, that is, with a global compilation cache, with or without compilation. The actual values were obtained by summing the ones collected individually for each library, but they confirm the global timings reported at the end of section 3, which were obtained in another run of the scenarios. Namely, compilation takes 16m 49s, loading requires 03m 22s, Declt needs 43m 19s, and Makeinfo runs for 43m 06s. Note that the only measured redundancy here is in ASDF load times. Indeed, compilation, Declt, and Makeinfo processing occur only once per library. However, the load time measurements include not only the libraries themselves, but also their dependencies, such that the actual measure for one library corresponds to the number of times it appears as a dependency, plus one.

[9]https://github.com/didierverna/quickref-benchmarks

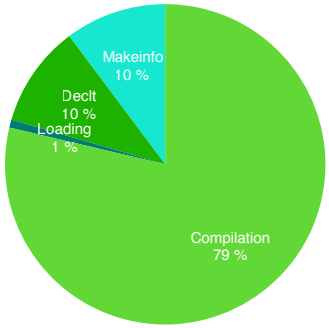

Figure 4: Time Distribution w/ separate compilation

The important information we get is that Declt and Makeinfo require practically the same amount of time to run in total. By summing ASDF time and Declt time, we see that in scenario 1 (no compilation required), Texinfo generation takes 52% of the time, versus 48% for HTML generation. In scenario 2, Texinfo generation takes 60% of the time, versus 40% for HTML generation. We did not collect numbers for scenario 3 (separate compilation directories for every library), but we can reconstruct them quite easily. Indeed, the Makeinfo, Declt, and ASDF load times are the same. The remainder of the 7h 01m, which amounts to 5h 32m, is thus devoted to (redundant) compilation. In this scenario, shown in Figure 4, Texinfo generation takes 90% of the time while HTML generation involves only the other 10%.

4.2 Time Shapes

Figures 5 and 6 provide two different views on the Declt processing real times. The first one displays the timings on a logarithmic scale, library per library (the libraries appear by lexicographic order on the X axis). The second one provides a histogram of the same data, with logarithmic scale on both axis. The actual numbers unimportant. What is important, on the other hand, is the general shape and characteristics of the data distribution.

The first remark is the very wide range of processing times. They spread from approximately 1s to 5m 19s. The second remark is that

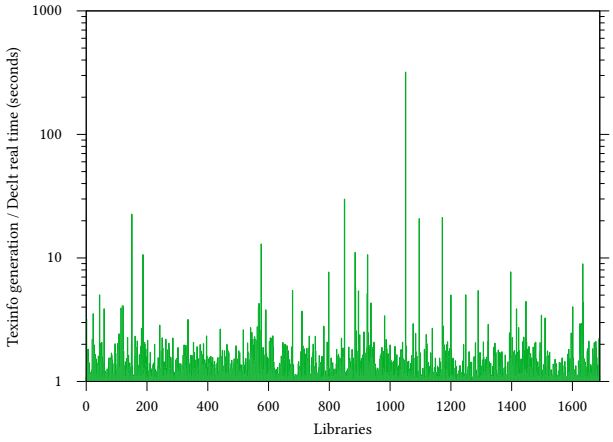

Figure 5: Declt real time per library

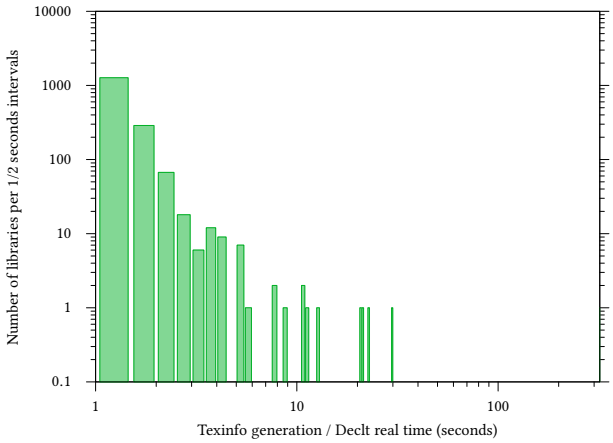

Figure 6: Declt real time histogram

most of the processing times are short compared to the maximum value. 75% of the libraries are processed in less than 1.5s, 92% in less than 2s, and 96% under 2.5s. Only 20 libraries require more than 5s for processing. Unfortunately, and this is the third remark, we cannot really take advantage of this knowledge in our parallel design, because there isn't any actual probability law underlying this data distribution. The average time is 1.67s, the median is of 1.23s, but the standard deviation is 7.99, which is meaningless, given the fact that all our timings are positive. In fact, the reason for this is that we cannot discard the "aberrant" values as experimental accidents, because they aren't: they *are* reproducible. For example, the library taking more than 5 minutes of Declt processing is `lisp-interface-library`. Further investigation shows that no matter how many times we repeat the Declt run, the timing *will* remain within that order of magnitude. Thus, when a thread is busy running Declt on that library, it *will* stay busy for around 5 minutes, a time during which 180 average libraries could be processed. That is 10% of Quicklisp!

We have conducted the same analysis for ASDF compile & load times, ASDF load times only (pre-compiled), and Makeinfo processing times. We do not report the results here. Suffice to say that in every case, we note the same kind of morphology: very wide range of values, high concentration of smaller values with a small, yet undiscardable number of "aberrations".

5 PARALLEL SOLUTIONS

As of this writing, the parallel solutions presented below are all implemented in a Quickref subsystem, automatically loaded when SBCL has multi-threading support compiled in. They may be dynamically selected and parametrized (*e.g.* number of threads for each task) through a set of keywords to the main Quickref entry point (the `build` function).

5.1 Solution 1

The first proposed solution is presented in Figure 7. This solution uses only two threads, and takes advantage of the natural sequencing of operations to establish a shared buffer of Texinfo files between Declt and Makeinfo. The main thread builds the Texinfo files sequentially (in any order). The second thread waits for them, grabs them (possibly by batches, emptying the shared buffer in one shot), and converts them to HTML.

5.1.1 Advantages. This solution is very simple to implement. There is only one shared resource: the buffer of Texinfo files. Only two threads are required, so it can work on older CPU's (*e.g.* dual-core without hyper-threading), or be less demanding on an otherwise busy or shared computer. Because the libraries are processed sequentially by Declt, no concurrent compilation occurs, so this solution may be used in either of our three scenarios.

5.1.2 Drawbacks. This solution's strengths flow from the same well as its weaknesses. Because only two threads are used, it will not take full advantage of the available resources. Besides, we know from Section 4.1 that depending on the scenario, Texinfo processing takes 48%, 40%, or only 10% of the time. This means that the HTML thread will in general be waiting more than working.

5.1.3 Experimentation. Experimentation with this algorithm confirms our analysis. Scenario one (no compilation) now takes 48m 30s instead of 1h 27m (almost twice as fast). Scenario 2 (global cache policy) takes 1h 05m instead of 1h 51m (we save roughly 40% of the time). Scenario 3 (local cache) takes 6h 22m instead of 7h 01m (we save around 10% of the time).

Note that because the Texinfo files are completely independent of each other and have no dependencies, it is straightforward to add more threads for Texinfo processing (the exact same function may be spawned multiple times). This, however, would be useless, as Texinfo processing is *not* where most of the time is spent. Again, experimentation also confirms this. Only in the next solutions will it become profitable to parallelize HTML generation.

5.2 Solution 2

Solution 2, presented in Figure 8 is a logical extension to solution 1. This time, the main process spawns several threads building Texinfo files in parallel, and several others generating the HTML ones. As

Figure 7: Solution 1
Main thread, Html thread

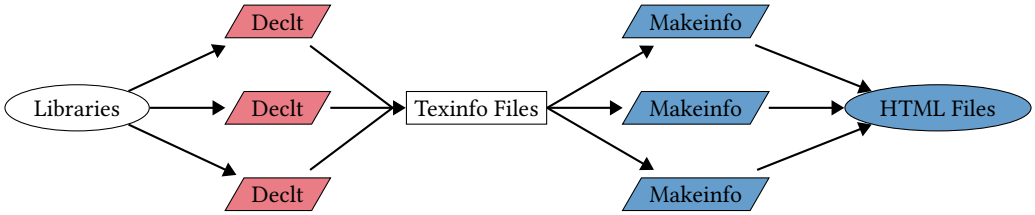

Figure 8: Solution 2
Declt threads, Html threads

before, a shared buffer of Texinfo files is used, but compared to solution 1, there are some notable differences or complications.

- Because multiple Declt threads exist, the original pool of libraries now becomes a shared resource. The Declt threads must hence synchronize on it as well as on the shared buffer of Texinfo files.
- Contrary to solution 1, the Makeinfo threads must *not* empty the buffer at once, grabbing a whole batch of Texinfo files to process. Indeed, we have learnt from Section 4.2 that the time distribution is not homogeneous, and that some libraries take an extremely long time to process, compared to the average. Thus, if we were to grab batches of Texinfo files, we would risk an *accumulation effect*, whereby multiple "short" Texinfo files would be blocked behind a "long" one, essentially re-sequentializing Html generation.
- For the exact same reason, it would seem ill-advised to simply split the initial pool of libraries into as many Declt threads as there are, and let them process their own batch sequentially. Instead, each thread will just process one library at the time.

5.2.1 Advantages. This solution will let us fine-tune the number of threads devoted to each task, depending on the machine at hand, or the scenario involved.

5.2.2 Drawbacks. Because the libraries are processed in parallel by Declt, in no particular order, concurrent compilation of common dependencies may occur. This solution can thus be safely used in scenarios 1 and 3 only.

5.2.3 Experimentation. Fortified by the time distribution reported in Section 4.1, we were able to fine-tune the number of threads in this solution to match our expectations. For scenario 1 (no compilation), the best results are achieved with the same number of threads for Declt and Makeinfo, specifically 4 and 4, corresponding to the hyper-threaded quad-core hardware configuration used in the experiments. It now takes 21m 47s to complete scenario one, which corresponds roughly to 25% of the original 1h 27m. For scenario 3 (local cache), the best results are achieved with 8 Declt threads and 2 Makeinfo threads. It now takes 1h 51m to complete scenario 3, which corresponds roughly to 26% of the original 7h 01m.

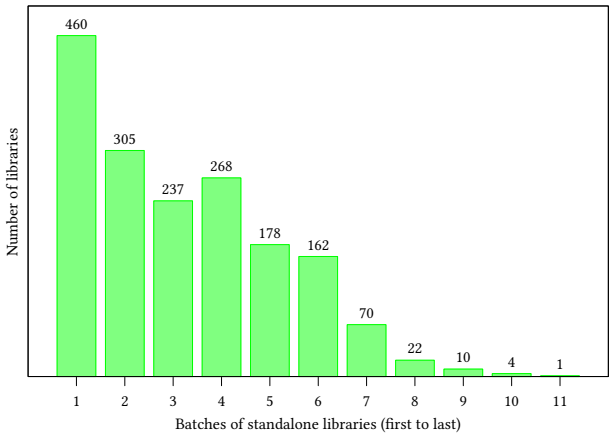

Figure 9: Library batches

5.3 Solution 3

Solution 3 is a refinement of solution 2 aiming at making it work with scenario 2 (global compilation cache). Remember that the complication comes from two libraries sharing common dependencies. Any attempt at loading them in parallel could result in the simultaneous compilation of the same dependency, followed by concurrent writing of the same *fasl* file. In order to prevent this, we must ensure that libraries processed in parallel by Declt do not have any dependencies in common, or only already compiled ones. We call those *standalone* libraries.

This problem is of course closely related to that of topological sorting[6], with the exception that we don't need full serialization. On the contrary, we want to retrieve batches of standalone libraries for parallel processing. The proposed solution is quite simple. First, we build a dependency graph of the libraries. The leaves in this graph do not have any dependencies, so they can be processed in parallel. We collect them; they constitute our first batch. We remove them from the graph, which leads to a new set of leaves, constituting the second batch. We repeat this process until the graph is exhausted. For the curious, Figure 9 shows those batches

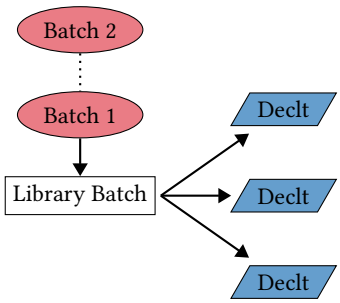

Figure 10: Solution 3, stage 1
Main thread, Declt threads

in the current Quicklisp distribution. We got 11 batches of 460 to only 1 libraries, from first to last.

In order to adapt solution 2 to this new scheme, a new shared buffer is created (see Figure 10). The Declt threads pick libraries from it instead of from the original libraries pool. The main thread sends successive batches of standalone libraries to this buffer, and waits for them to have been exhausted before sending the next batch in. The rest of solution 2 is unchanged (in particular, the HTML generation code can be re-used without modification).

5.3.1 Advantages. At the expense of a slightly more complicated synchronization logic, this solution may be used in any of our 3 scenarios. In the current status of Quicklisp, the dependency graph is relatively small (less than two thousand nodes), which means that the additional computation time required to handle it is negligible compared to the 21m 47s of our current most optimistic situation.

5.3.2 Drawbacks. Before sending the next batch in, the main thread must wait for *all* libraries in the current batch to have been entirely processed by a Declt thread; not just have been picked up by one of them. At a first glance, this may not appear as a serious issue because we only have 11 batches and a few threads handling them. However, remember again from Section 4.2 that some libraries *will* take a very long time to process. If, for example, such a "long" library is part of a small batch, the batch will be quickly emptied, and all Declt threads will essentially become dormant until the "long" library is treated. This is yet another form of *accumulation effect* that can potentially hinder the parallelization.

5.3.3 Experimentation. Because the time required to maintain the dependency graph is negligible, this solution is not expected to make much difference in scenarios 1 (no compilation) and 3 (local cache), as it would boil down to handling the libraries in a different order. For scenario 2, the best result was obtained with an equal number of threads for Declt and Makeinfo, namely, 4 of each (again, corresponding to the hyper-threaded quad-core hardware configuration used in the experiments). There, the overall computation time fell down to 29m 21s, that is, 26% of the original sequential time. Given the time distribution in Figure 2, we also tried matching that proportion, for example with 5 Declt threads and 3 Makeinfo ones. We only got similar (inconclusive) result only differing by less than 5%.

6 CONCLUSION

As mentioned in the introduction, the absolute worst case scenario for Quickref, which is to build the complete Quicklisp documentation from scratch, takes around 7 hours on our test machine. Even if such a duration may appear reasonable for batch processing, we still believe that parallelization is not a vain endeavor. First of all, the ability to use Quickref interactively (creating for example one's own local documentation website) makes it worth improving its efficiency as much as possible. Secondly, Quicklisp itself is an ever-growing repository (monthly updates usually add at least a dozen new libraries to the pool), and so is the time to generate the documentation for it.

In this paper, we have devised a set of parallel algorithms, and experimented with them in different scenarios corresponding to the typical use-cases of Quickref. On our test machine, we were able to reduce the required processing time roughly by a factor of 4 compared to the naive sequential version, which is already quite satisfactory. The absolute worst-case scenario fell under 2 hours, and the most frequent one under half an hour. For all that, and in spite of the fact that gracefully handling concurrency is always a tricky business, our parallel solutions remain quite simple. The implementation of solution 3, for example, requires only 3 shared resources (2 buffers and a counter), 2 mutexes and 3 condition variables. It was implemented directly with SBCL's multi-threading layer, without resorting to higher level libraries.

This work also lead us to perform various preliminary measurements and analysis on Common Lisp libraries (compilation and load time, Declt and Makeinfo run time, dependency graphs, *etc.*). As mentioned before, the collected experimental data and their interpretation is publicly available. We think this data could be useful for other projects, and we already know for a fact that the current Texinfo maintainers are interested. Only a small part of those results have been presented in this paper. We are confident the rest will be extremely useful for future refinements. Indeed, there are still many things that can be done to improve the situation even more.

7 DISCUSSION & PERSPECTIVES

7.1 Alternative Solution

Yet another, alternative, parallel solution exists, depicted in its entirety in Figure 11. This solution consists in processing the libraries in parallel, yet, without breaking the Declt / makeinfo chain. Multiple threads (8 would probably be an appropriate number on our test machine) pick libraries to process, and sequentially run Declt followed by Makeinfo on them. As solution 2 (Section 5.2), this algorithm can be made to work on scenarios 1 and 3 only, or, as solution 3 (Section 5.3) can be combined with library batches in order to also work on scenario 2. This is what Figure 11 depicts. In the future, and mostly out of curiosity, we may experiment with this solution.

Note however that we don't expect it to make much difference compared to solution 3. In solution 3, we have indeed fewer threads picking libraries up for Declt processing, but on the other hand, these threads also return more quickly to the library pool / batch, since they are not in charge of Makeinfo. In fact, our gut feeling is

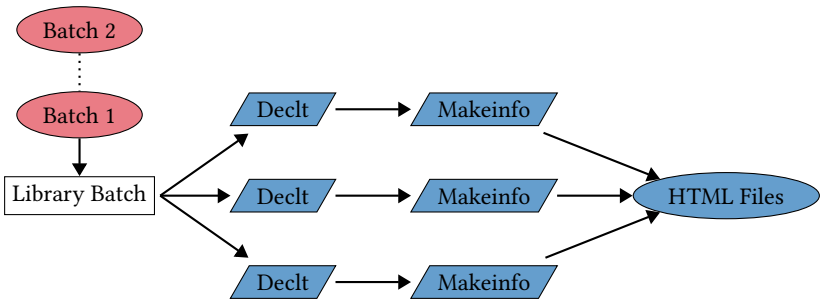

Figure 11: Alternative Solution
Main thread, Declt & Makeinfo threads

that solution 3 may remain slightly better, as it is probably more gentle on the overall waiting times.

7.2 Dependency Management Issues

Dependency management, required by solution 3, is a relatively fragile mechanism. Currently, we base our knowledge of dependencies on static information provided by Quicklisp directly, more specifically, the `required-systems` slot from the `ql-dist:system` class. This information is based on ASDF's `depends-on`, `defsystem-depends-on`, and also comes from observing the state of the environment before and after loading the library.

The reliability of that information is somehow relative, however. Any inaccuracy in that information can potentially lead to a corrupted dependency graph, in turn risking unprotected concurrent compilation. Here is, for example, one such scenario, reported by the author of Quicklisp. This problem is currently known to affect a couple of libraries.

> *Consider systems A and B, where A requires B to build. When Quicklisp test-builds A, the A prerequisites are built in such a way that B also successfully builds to satisfy A's requirements. But when Quicklisp test-builds B on its own, the environment is different in a way that precludes B from building. In that case, the metadata in Quicklisp specifies that A requires B, but B is not listed at all, because it does not build on its own.*

7.3 Library Ordering Refinements

In Section 5.3, we introduced the idea of *library batches*, that is, sets of libraries the loading of which wouldn't entail any compilation conflicts, and we mentioned the need to wait for batch exhaustion before sending in the next one. This requirement, which *is* a limitation, actually comes from the fact that only static information (namely, the dependency relations) is used to create the batches in question.

It is however possible to refine this idea. Indeed, the most pertinent information for us is *not* that library 1 depends on library 2, but that the compilation of library 2 is over. In other words: concurrent compilation is problematic; concurrent loading is not.

In order to improve on solution 3, we hence need one additional piece of (run-time) information: we need to be notified when a Declt thread has finished processing a library. The refinement can then go as follows. We create the same dependency graph as before, and also initialize the library queue with the first batch as before (the libraries with no dependencies), but this time, *without* removing them from the graph. From now on, as soon as a library is done processing by a Declt thread, we remove it from the graph. Any *new* leaf in the graph stemming from that removal can then safely be pushed immediately to the library queue.

An even better refinement would be to *not* wait for Declt to finish processing, but only for ASDF to finish compiling (this would require a communication channel between the main thread and the external Declt process though). This refinement has not been implemented yet, but it is a high priority, as we expect a somewhat substantial gain from it. Note also that it can be used in the alternative solution proposed in Section 7.1.

Currently, our dependency graph is implemented as a hash table of *adjacency lists*[3]: the hash keys are the library names, and the hash values are the lists of dependencies. Another possible (and classical) implementation consists in using an *adjacency matrix*[1], a potentially more compact representation. Whether one representation would be more beneficial than the other is currently unknown. In particular, more investigation on the dependencies morphology should be conducted, notably to discover whether the adjacency matrix would risk being sparse or not (very likely). In any case, the choice of representation is not expected to have much impact on the performance, again, because the dependency graph is relatively small (less than two thousand nodes), in front of at least 20 minutes of total processing.

7.4 CPU *vs.* I/O Consumption

While the performance improvements obtained from solution 1 are to be expected, getting *only* an improvement factor of 4 in solution 2 or 3 may appear somewhat surprising, even disappointing, especially since our test machine has 8 virtual cores (4 hyper-threaded actual cores). Of course, a factor of 8 would be unrealistic. Studies (from Intel or otherwise) have shown that an improvement of 30% is not unreasonable to expect from hyper-threading[2, 11]. The problem we have here is the fact that both Declt and Makeinfo are treated as monolithic black boxes, so we don't have any control on their CPU *vs.* I/O consumption, an otherwise important aspect of parallelization.

Declt works in two stages: first, an abstract in-memory representation of the documentation is constructed by introspecting the library. Next, the Texinfo file is generated from that abstract representation. The first stage is CPU-intensive, the second one is

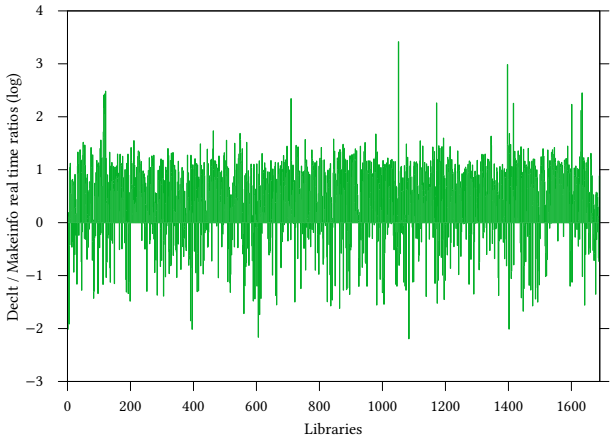

Figure 12: Declt / Makeinfo comparative timings

more pressing on I/O. On top of that, remember that the library needs to be loaded by ASDF first, possibly with some compilation. This will also entail several CPU or I/O intensive phases.

It turns out that Makeinfo works in a similar fashion, at least for HTML production (for PDF, TEX is used). The first stage reads the Texinfo file into an abstract in-memory representation. This stage is written in C, with a Perl interface. Then, the abstract representation is altered in various ways, and the HTML file is finally created. This last stage is entirely done in Perl, and (according to the current Texinfo maintainers) is probably much slower than the previous ones.

Having no control over these different processing phases is unfortunate, and is likely to be the cause of the "25% threshold" that we seem to reach. It may very well happen, for instance, that regardless of the number of threads that we have, they all end up in an I/O phase at the same time, essentially waiting on the same disk, while subsequent CPU-intensive computation could have been started. Improving that situation would require cracking the Declt and Makeinfo "black boxes" open, possibly even the ASDF one, in order to introduce parallelism at a lower level. Although we already have some ideas about this, it would be a completely different project.

7.5 Scheduling

In addition to the points raised in the previous sections (the idea of library ordering in particular), the general question of scheduling could be raised. For example, we could get inspiration from the operating system theory, and think of improving things by minimizing the waiting time in the various queues, as the SJF (Shortest Job First) does in process scheduling[8]. The problem here is to get a notion of what makes for the complexity (hence the time) of the various tasks. Very preliminary investigation gives a somewhat pessimistic impression. For example, the collected experimental data shows that there is no correlation between Declt and Makeinfo processing time (see Figure 12). For some libraries, Declt takes more time than Makeinfo; for some others, it is the other way around, *etc.* In the worst case scenario, we would need to run Quickref and collect the data in question (which we did for this article), and use it on the next Quicklisp release, hoping that the situation didn't change too much.

7.6 SSD Technology

Finally, note that the validity of the present study is highly dependent on the fact that our test machine was equipped with a traditional, mechanical hard drive. We haven't had the opportunity to experiment with SSD (Solid State Drive[4]) technology yet, but their dramatically quicker access time and lower latency[5] is very likely to redefine the parameters of our study.

ACKNOWLEDGMENTS

The author would like to thank Zach Beane for his feedback on how Quicklisp handles dependencies across libraries, and Karl Berry, Gavin Smith, and Patrice Dumas for their insight into the inner workings of the Texinfo system.

REFERENCES

[1] Norman Biggs. *Algebraic Graph Theory*. Cambridge University Press, 1993. Definition 2.1, p. 7.
[2] Shawn D. Casey. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6(1), 2011.
[3] Thomas Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.
[4] Neal Ekker, Tom Coughlin, and Jim Handy. An introduction to solid state storage. SNIA White Paper, January 2009.
[5] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. Dell Technical White Paper, May 2011.
[6] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968. Section 2.2.3.
[7] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.
[8] Andrew S. Tannenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4th edition, 2014. Section 2.4.
[9] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.
[10] Didier Verna. Standard i/o syntax, and the robustness principle. https://www.didierverna.net/blog/index.php?post/2017/10/27/Standard-IO-syntax-and-the-Robustness-Principle, November 2017. Blog Entry.
[11] Duc Vianney. Hyper-threading speeds linux. https://www.ibm.com/developerworks/library/l-htl/index.html, 2003.