Improving Parallelism in Git and GCC: Strategies, Difficulties, and Lessons Learned

1st Matheus Tavares Bernardino
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
matheus.bernardino@usp.br

2nd Giuliano Belinassi
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
giuliano.belinassi@usp.br

3rd Paulo Meirelles
São Paulo School of Medicine
Federal University of São Paulo
São Paulo, Brazil
paulo.meirelles@unifesp.br

4th Eduardo Martins Guerra
Computer Science Faculty
Free University of Bozen-Bolzano
Bolzano, Italy
guerraem@gmail.com

5th Alfredo Goldman
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
gold@ime.usp.br

***Abstract*—Manufacturers are creating powerful CPUs by exponentially increasing the number of cores over time, as producing faster sequential chips has become more expensive. Developers must now employ parallel strategies and design parallel algorithms if they want to use every resource available in the machine. Still, many successful open-source projects are mostly sequential, failing to harness the full computational power available. This article presents approaches for performance improvements into two large and well-known open-source projects, Git and GCC, using parallel programming. We share the difficulties faced and the strategies used, concluding with a set of lessons learned useful to similar parallelization processes.**

***Index Terms*—parallelism and concurrency, parallel systems, compilers, version control, GCC, Git, conversion from sequential to parallel forms**

I. Introduction

Multicore computers dominate both consumer-level and server markets nowadays. Therefore, if developers want to use every resource available in modern machines, they will have to design and employ parallel algorithms in their projects. However, many successful projects still rely on sequential code or not wholly benefit from the parallel power available. This article reports the strategies used to improve or introduce threading in two large open-source projects: GCC and Git. Our main contribution is identifying the challenges faced and discussing the lessons learned, which might help in future parallelization processes of other open-source software.

II. Git Case Study

Git is an open-source version control system used to manage a wide variety of projects of different magnitudes. Therefore, high performance and scalability are some of the main priorities for the development community. With this in mind, the git-grep command was parallelized in 2010 using a producer-consumer mechanism. This command searches lines matching a given pattern in the files of a repository managed by Git. It can search both in the working tree, where the current project version is checked out, and in the internal object store, which contains older versions of the files.

With the 2010 conversion, the multithreaded git-grep achieved good speedups on the working tree. However, the parallel version turned out to be slower than the sequential one for object store grepping. Therefore threads were later disabled for this case. In this work, we sought to understand what caused the slowdown and improve git-grep parallelism, to re-enable threads in the object store case with satisfactory performance.

A. Git Objects

First, we analyzed how objects are stored and read, to understand why git-grep could not benefit from threading, when searching in them. Conceptually, Git's data store can be visualized as an in-disk key-value table. Each Git object is stored in a compressed binary format – using the zlib implementation of the DEFLATE algorithm [1] – and referenced by its SHA1 hash.

The three most simple types of Git objects are `blob`, `tree`, and `commit`. The first stores a generic stream of bytes, and it is commonly used to save the contents of files tracked by Git. The second is a table, storing the entries of a directory tree. It references `blobs`, for file entries, and other `trees`, for subdirectories. Finally, the `commit` object is used to represent *versions* of a project. It contains a reference to a `tree` object, which describes the repository's state at that given version.

Occasionally, Git will group objects into a `packfile`, which is a very memory-efficient representation that holds many objects in a single indexed file. In this format, besides being zlib compressed, objects are also allowed to be deltified. I.e., two similar objects are not stored redundantly, but instead, a single base object is kept together with a set of instructions on how to reconstruct the other.

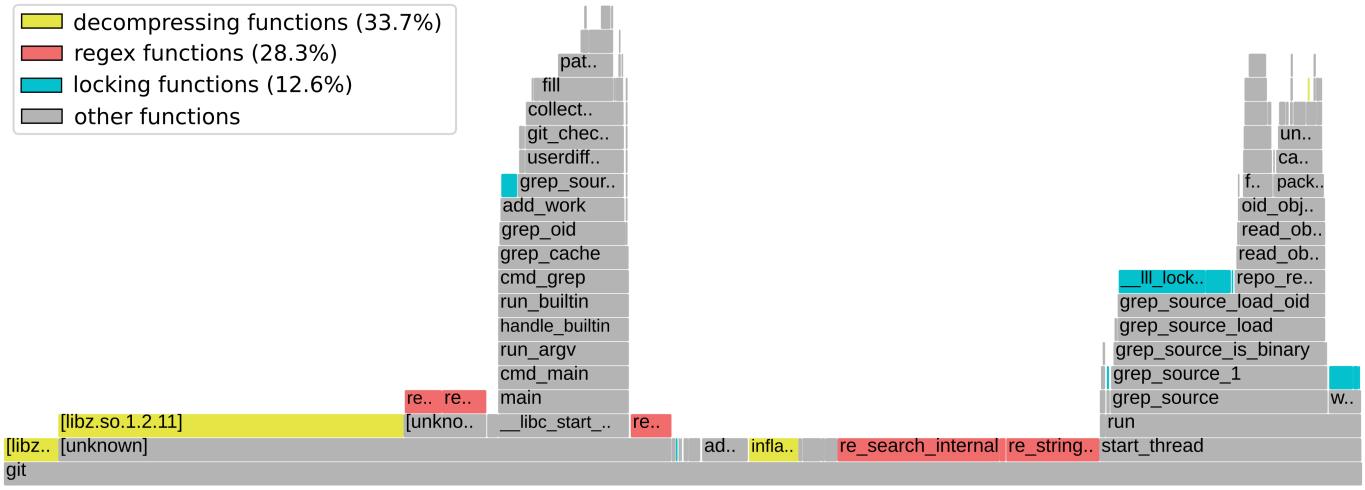


Fig. 1: Visualization of `perf` profile for a `git-grep` execution in the object store, using chromium repository as data, with 8 threads.

B. Profiling and Locating Hot Spots

Object reading can be quite a complex task. Besides the actual disk reading operation, there are also packfile indexing, zlib decompression, and delta reconstruction. Furthermore, the object reading machinery wasn't thread-safe, and thus, required serialized access among multiple threads. To verify how object reading contributed to the grep slowness when threaded, we profiled the code with `perf` (https://perf.wiki.kernel.org). We compiled Git with `-O0 -g` to append the necessary debugging information and disable compiler optimizations, since it results in a profiling output more faithful to the actual source files and, therefore, easier to understand. We used the Chromium repository as data (at commit `03ae96f` of 04-06-2019), for being relatively large in both content and history. We also used the following extended regex for being a practical use case and producing a quite time-demanding search:

```
(static|extern) (int|double) \*
```

We performed our experiments in a desktop running Debian 10.0 on a Intel(R) Xeon(R) CPU E3-1230 V2 (4-cores with hyper-threading), 32GB of RAM and a 7200 rpm, SATA 3.1 HDD. Figure 1 shows the results using `FlameGraph` [2] for visualization. Each rectangle represents a stack frame; the larger the rectangle, the more time the process spent on it. We also colored the figure to highlight some sets of functions of interest. As we can see, decompression alone accounted for up to one-third of git-grep total execution time.

C. Improving Parallelization

Being responsible for a significant portion of the execution time, we focused our efforts on the decompression operation. We first moved the mutex that had been used to protect calls to object reading functions in git-grep down to the actual object reading functions, protecting them from the inside. Then, we released the mutex only during decompression, to execute it in parallel. The actual implementation, however, involved many nuances: just by opening a single hole in this critical section, we risked having global resources modified between the execution of the first and the second portions, which could produce race conditions.

We faced this problem with the delta base cache, which optimizes delta reconstruction by holding previously decompressed bases in memory. When there is a cache miss, the code flow executes decompression and inserts the result to the cache. However, insertion is done without re-checking if the entry is still absent. This was not a problem for the single-threaded execution, but the second check is required when we allow decompression to run in parallel, to avoid inserting duplicated entries.

There were also other difficulties, such as the previous git-grep mutex protecting function calls in different abstraction layers; and the exposure of unprotected lazy initializers when moving the lock down in the stack. To solve the former, we turned the internal object reading lock into a recursive mutex and exposed it for external use, without the risk of relocking errors. About the latter, we protected the more accessible initializers and forced an eager initialization on the two more difficult to protect. This technique should be used cautiously as it can potentially diminish performance for the cases where the resource is not necessary.

D. Additional Improvements

We also worked towards allowing more parallel regions besides decompression in two places: the functions that handle git submodules in git-grep; and a code snippet inside the producer-consumer critical section. Regarding the latter, we moved an already thread-safe function call out of the mentioned critical section. Although small, this change brought significant time reductions, even for the working three cases of git-grep.

E. Validating Correctness

To evaluate correctness, we used several different tests and tools. The major being Git's test suite, which contained seven test files for git-grep, totalling 424 individual tests. Together they covered about 96% of the code lines added in this work (117 from 121).

The test suite successfully exposed problems originated from incorrect refactoring and initial threading errors. However, it did not always report trickier synchronization problems. These only appeared with heavier loads, which are not very suitable for the common test base, as they significantly increase runtime. So when the most frequent race conditions were fixed, we started testing with larger real-word repositories, such as Linux and Chromium, together with tools like `helgrind` and `memcheck`. This was essential to find the problem of duplicated entries in the delta base cache.

III. GCC Case Study

The GNU Compiler Collections (GCC) is widely used due to its matureness, reliability, performance, and extensions provided to the C/C++ languages. Although LLVM compilers are getting more attention from academics nowadays, GCC still supports more hardware architectures.

Compiler parallelism has already been studied before [3], [4], but these studies are rarely applied in realistic compilers. More commonly, parallelism is achieved through Makefile rules that allow the compilation of multiple files simultaneously. Additionally, GCC Link Time Optimization (LTO) also employs parallel processing after the Whole Program Analysis is complete through multiple processes, but is often slower than the classical per-file compilation [5].

The discussion about reproducible builds is also beyond our scope, and future studies are required for this subject.

A. Compiler Optimization

Profiling reports indicated that optimization was the most time-consuming part. To understand how optimizations perform, we can break the set of all optimizations into two disjoint subsets:

- *Intra Procedural Analysis* (IraPA) only uses information found within the function. Example: vectorization.
- *Inter Procedural Analysis* (IPA) analyses how functions interact with each other. Example: function call inlining.

Later, profiling showed that IraPA consumed 75% of compilation time with *gimple-match.c*, a file generated from the compilation of "match.pd". This file was chosen for our tests because it's the largest in the GCC project, with 113207 lines, and it evidences a bottleneck issue [6]. We are focused on files of this magnitude, as they will benefit the most of our parallel implementation. Later, we plan to evaluate the minimum file size for which it is worth running the parallel version and switch to the sequential code for smaller files.

In GCC, IraPA are performed on two distinct Intermediary Representations (IR): GIMPLE and RTL. GIMPLE is a hardware-independent IR [7]. However, RTL is a hardware-dependent representation that is as close as possible to the hardware; resulting in specific code for each target architecture GCC supports.

So, our primary focus was on parallelizing GIMPLE optimizations using threads. We chose GIMPLE as it is hardware-independent, improving every target supported by GCC and because the community showed interest in this research. Our results about GIMPLE can also be used to estimate the improvement when RTL is also parallelized.

B. Improving Parallelization

We placed a producer-consumer queue between IPA and IraPA to parallelize these optimizations, as previous works on compiler parallelization [4]. Thus, when IPA finishes analyzing, the threads can pop functions from this queue and schedule them to optimize in parallel.

GCC was initially designed to compile entirely one function at a time, and some data structures that requires replication for each function were represented in a singleton. This choice was made back in early GCC development, where computers had a minimal amount of memory. However, This lack of structure replication is painful when inserting parallelism into legacy code. To fix these issues, we ensure that all GIMPLE optimizations were applied to every function before performing RTL optimizations, and ensured that all automatic tests passed before continuing. This ensured that a first set of singleton structures were replicated.

After this queue was implemented and functions were compiled in parallel, several race conditions showed up. However, these issues can be fixed with time and effort because IraPA optimizations should not interact with other functions [8].

C. Memory Management

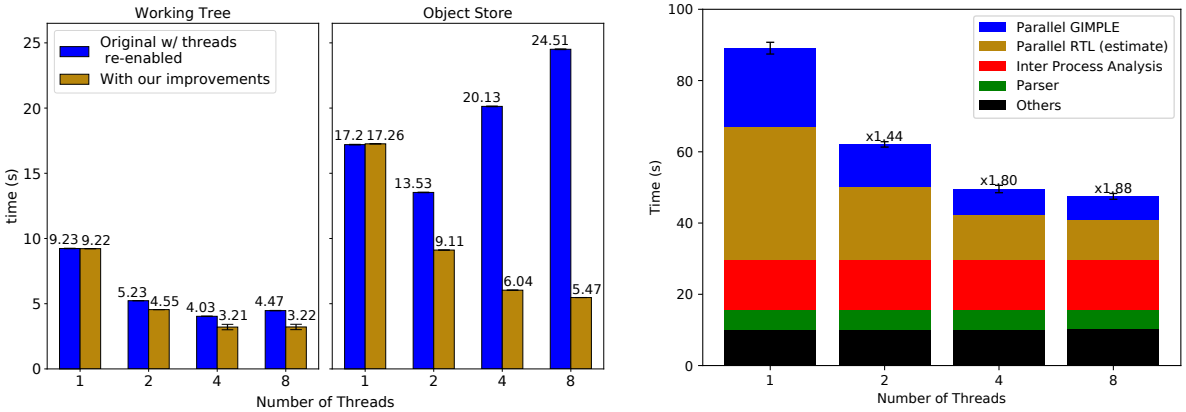
The first issue we tackled was memory management. GCC uses memory pools to avoid repeated calls to `malloc` and `free`, as well as to memory alignment to speed up accesses. Instead of protecting the linked-lists implementation, we used a distributed approach where each thread has its private pool. This strategy does not require locks and therefore is significantly faster, but each pool becomes private to its thread, resulting in another problem: data required later in the compilation were lost when threads are joined. Therefore we implemented a feature that merge the pools right before the thread finalizes, but only executing when necessary. Finding the required pools to be merged was not complicated due to GCC extensive test suite.

Another memory-related issue was garbage collecting. GCC has an internal garbage collector that collects unreferenced objects when called. Currently, we serialized this feature using a single mutex and disabled collection between optimization passes, as the Garbage Collector global states must still be mapped.

D. Validating Correctness

Our parallelization effort in GCC is a work in progress, so there are still concurrency issues that must be solved. We currently have 99.8% approval rate for `gcc.dg`'s testsuite (22 failures from 13183 test cases).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2020.3020932, IEEE Software



(a) Time comparison between original and final git-grep code

(b) GCC Time and Speedups (Estimate)

Fig. 2: Timing results.

More complex synchronization errors were hard to detect with the testsuite, so we used load tests together with synchronization error detectors. However, there is a trade off: increasing the input size helps exposing unusual errors, but it also slows down the runtime, especially under debugging tools. Additionally, to further increase the exposure of uncommon errors, we manually forced extreme conditions in the code, by introducing synchronization barriers right after each pass.

Currently, self-compilation of the C frontend is working; but race conditions prevented libstdc++ from compiling. These results encouraged us to try another approach to the same problem, using processes instead of threads.

IV. Results

Concerning the Git case study, the code changes proposed in our work were applied to the upstream project and released as part of Git version 2.26.0. The individual patches can be seen at: https://patchwork.kernel.org/project/git/list/?series=228919.

We present the timing results in Figure 2a, which contains the mean elapsed times of git-grep in intervals with 95% of confidence. This plot compares the execution times of the original code to the code after our improvements. Note that the original code did not allow threads for object store, but we enabled them for comparison. All tests were performed using the methodology described in Subsection II-B, using the chromium repository for testing data and searching the mentioned regex for C code. However, this time each test was repeated 30 times after two warmup executions to populate the caches.

From the observed results, we can conclude that the proposed changes successfully increased git-grep performance in the object store case. The speedup was up to $3.14\times$ over the original code and $4.5\times$ over the threaded code without our improvements. Additionally, we saw a time reduction of almost 28% in the working tree case. With further investigations, we discovered that this speedup came from the thread-safe code that was moved out of the producer-consumer critical section.

Regarding the GCC case study, our approach produced a preliminary speedup in GIMPLE up to $3.35\times$ with eight threads. It produces a speedup of up to $1.88\times$ in the entire compilation process when projected to all IraPA passes, as illustrated in Figure 2b. We executed our experiments in the same machine as the Git case, using the *gimple-match.c* file.

V. Challenges

Git and GCC are both large projects with big codebases. So it can take a while for new contributors to get reasonably familiar with the code and its interfaces. Furthermore, tasks such as parallelization usually involves going through many call chains and analyzing the effect of the operations on global states. Good knowledge of the code is indispensable in this context. Therefore, studying the codebase was one of the first significant challenges.

During the process, we also encountered difficulties regarding the usage of techniques that are not thread-safe. The next two subsections describe some of these techniques used in Git, and the third describes a technique used in both Git and GCC.

A. Lazy Initializers

Lazy initializers prepare resources on demand, avoiding the initialization of variables that will not be needed in a particular execution. In comparison to preinitializing variables during program startup, a drawback of lazy initializers is that they must be protected if they might run in threaded sections. Otherwise, two or more threads might try to initialize the resource simultaneously, leading to data corruption.

Since some lazy initializers in Git were not used in threaded code before, we had to protect them. But in two cases, where the initializers were less accessible, we decided to execute them eagerly before spawning threads, avoiding to add more locks in the parallel sections. As previously mentioned, such a technique can incur additional overhead in cases where the

resources are not needed. To avoid this problem, we (1) only perform eager initializations in multithreaded mode and (2) try to evaluate, when possible, if the resource will be needed before initializing it.

B. Function-Scope Static Variables

In the C language, function variables declared as static are not destroyed when the function returns. Instead, they persist throughout the whole execution. For this reason, they are often used in Git as returning data, since it excuses callers from the responsibility of releasing memory after using it. However, as these variables are shared among threads, two parallel calls to the function may corrupt the result of each other.

In git-grep, most of the function-scope static variables in threaded code were placed in lower level functions; therefore the higher-level object reading mutex already covered their use. Nevertheless, we had to be careful in refining this mutex, not to accidentally remove the protection around any function call that uses such static variables.

There are also lockless solutions to this problem. When the function-scope static variables are used solely to return data, not to share values with other threads, the function might have a separate variable for each thread (e.g., using Thread Local Storage). This allows more parallel work, at the cost of higher memory usage. In the case study, though, this strategy would probably not have a significant impact, as the places were we would apply it already required a lock, for other operations. Furthermore, the major performance bottleneck was not in these functions, but in the decompression calls.

C. Use of Global Variables in Different Abstraction Layers

The Git and the GCC codebases use some strategical global variables to avoid passing down the same data in many call stacks. Two examples are the Git `the_repository` variable, which holds information from the repository being operated, and the GCC `cfun`, which holds information about the current function being compiled. Nevertheless, the use of this kind of variable in different abstraction layers can sometimes complicate the process of evaluating thread-safety: one might erroneously think that a function is safe for not using global states while, in fact, it calls another function that does. Furthermore, this unsafe call may be deeper in the call tree, making it harder to locate.

In Git, we manually scanned the call trees with assistance from call graph generators to find and protect code paths that would lead to data races. We also used `ctags` to easily jump between symbols, and custom scripts to filter out known protected paths from the call graphs. This approach can be dangerous, as it relies on manual analysis, but we sought to reduced the risk of leaving thread-unsafe operations behind by using assistant tools and synchronization error detectors. Another alternative might be using `GCC with plugins` (https://gcc-python-plugin.readthedocs.io/en/latest/working-with-c.html#finding-global-variables).

In GCC, some global variables did not have to be shared among threads, so they could be made thread-local to avoid race conditions. However, because of the size and age of the project, finding the sources of all race conditions on other global variables revealed to be very difficult. Therefore we are now exploring the possibility of using processes instead of threads, which drastically changes the problem of finding race conditions into mapping which resources are required to be shared (through `mmap()` and FIFOs). Besides speeding up the development process, we also believe that this approach substantially reduces the risk of having uncaught race conditions in the final code.

VI. Lessons Learned

A. Community support is crucial

First of all, we highlight the importance of community interactions. The Git and GCC communities had a crucial role in both the planning and development phases, collaboratively proposing ideas and elaborating strategies with us. In some cases, developers have directly contributed with code, or prototyped concepts to test the effectiveness of the proposed plans. Additionally, the rounds of code review also significantly improved the quality and safety of our changes.

Finally, as part of the project was developed during Google Summer of Code, we were also paired with more experienced developers for direct mentorship. This kind of assistance is a great way to engage newcomers and help them progress to faster.

B. Take time to truly understand the bottleneck

Knowing what can and should be executed in parallel is already a big step. This statement might seem obvious, but it can be entirely overlooked in a rush to achieve better performance. Not all time-consuming operations can be parallelized, and not all parallelizable tasks will result in noticeable performance improvements. So it is essential to engage in a preparatory period to locate and understand the hotspots and avoid spending time on unsustainable paths. Furthermore, profiling tools, such as `perf` and `gprof`, can be used to assist developers in this task. This preparatory period might also include coding activity. Once the hotspots are located, it is advisable to implement a couple of ”quick and dirty” prototypes, to evaluate performance and feasibility. This approach helps evaluate ideas early, by predicting what speedup could be achieved through specific paths.

C. Threads are not the only way

When converting a sequential program to parallel, we typically first consider the use of threads, since data sharing and synchronization are much easier to deal using threads than subprocesses. However, when the original code has too many global and thread-unsafe resources, adding threads can be cumbersome, sometimes requiring a major code refactoring before introducing parallelism. In this case, process parallelization is an alternative to be considered, as the thread-unsafe variables are kept independently by default, and the programmer can manually and incrementally decide what is going to be shared. This approach is what we are currently

attempting in `GCC`. One of the downsides is the higher cost for data sharing, as it relies on inter-process communication. However, if communication is not so frequent, which is the case in GCC, that drawback can have an acceptable impact.

D. Refine mutex granularity just enough

Both coarse-grained and fine-grained locking strategies have their advantages and disadvantages. We usually achieve more parallelism with finer-grained locks, but with the cost of higher synchronization overhead. Additionally, the development is more error-prone, as it is easier to forget the different locking protocols before attempting to use the resources. Certainly, the ideal granularity depends on the problem being parallelized. Nevertheless, it is important to highlight that even small changes to the locking granularity can sometimes produce huge performance changes. This avoids the need of a much larger code refactoring to change the locking strategy completely.

This could be very clearly observed in `git-grep`: the code used to have a coarse-grained lock, protecting the whole object reading machinery. We did not refactor the code to remove this lock in favor of fine-grained ones. Instead, we refined it just enough to allow running decompression locklessly. Since this was the most time-consuming operation in object reading, the change resulted in an significant speedup.

E. Keep in mind the alternatives for synchronization

Novice parallel developers might tend to stick with the more obvious mutual exclusion locks to perform synchronization. Though various alternatives might be more or less suitable for each situation. For example, conditional variables, barriers, monitors, and the many types of locks: spinlocks, read-write locks, recursive locks, and others. In `git-grep`, the conversion from a common mutex to a recursive one allowed the lock refinement with fewer changes, as race sections in different abstraction levels could be protected with the same structure. Without a recursive lock, more extensive refactoring would be required to achieve a similar result.

F. Prefer a distributive memory approach

If possible, it is even better to avoid using synchronization mechanisms altogether. Try to find false dependencies and, in such cases, opt for a distributive memory approach rather than using mutexes. That strategy was used in the GCC case when handling very frequent operations such as memory managing through a memory pool. A centralized solution to race conditions, protecting the pool with mutexes, slowed down the compilation to the point that the speed test was failing. By removing the need for synchronization with independent memory pools for each thread we significantly improved our parallel implementation performance.

G. Complement a large test base with synchronization error detectors and load tests

Tests are essential for any code change, especially in a large refactoring, which is usually required for parallelization. Nevertheless, some multithreading problems appear rarely or only under specific circumstances, thus not being detected even by a large test base. For instance, despite the high acceptance ratio, GCC bootstrap was not working; and the duplicated cache insertions in Git were only found with larger repositories. This happens because test suites typically use small data, both to be fast and to provide a controlled environment for testing. However, this might not generate enough workload to expose more complex threading errors. Therefore, we advise complementing the test suite with load tests which make threads work longer. Additionally, developers can temporarily tweak the code to force extreme conditions that induce the manifestation of synchronization errors. This can be done adding barriers, disabling caches, etc.

Finally, we advise using synchronization error detectors – such as `ThreadSanitizer`, `Helgrind`, and `DRD` – and other debugging tools, as `memcheck`, which is particularly useful to identify issues where a thread releases variables required by other threads.

VII. Acknowledgements

The Coordination for the Improvement of Higher Education Personnel (CAPES, Brazil) partially funds this work.

References

[1] L. P. Deutsch, "Deflate compressed data format specification version 1.3," RFC 1951, 1996. [Online]. Available: https://rfc-editor.org/rfc/rfc1951.txt
[2] B. Gregg, "The flame graph," *Commun. ACM*, vol. 59, no. 6, 2016.
[3] M. T. Vandevoorde, *Parallel compilation on a tightly coupled multiprocessor*. Systems Research Center, 1988.
[4] D. B. Wortman and M. D. Junkin, "A concurrent compiler for modula-2+," *SIGPLAN Not.*, vol. 27, no. 7, p. 68–81, 1992.
[5] T. Glek and J. Hubicka, "Optimizing real world applications with gcc link time optimization," *ArXiv*, vol. abs/1010.2196, 2010.
[6] Martin Liška, "Pr84402: Gcc build system: parallelism bottleneck," 2018. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84402
[7] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *Proceedings of the GCC Developers' Summit*, 2003, pp. 171–179.
[8] A. Aho, M. Lam, R. Sethi, and J. Ullman, "Compilers: Principles, techniques and tools, 2nd editio," 2007.